MASTER EN SCIENCES INFORMATIQUES

Production de commentaires en lignes de composants réutilisables C++

Strouboulis, Nicolas

Award date:
1996

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame
de la Paix - Namur

Production de commentaires en ligne de composants réutilisables C++.

Nicolas STROUBOULIS

Mémoire effectué dans le cadre de l'obtention du diplôme de licence et maîtrise en informatique.

Promoteur : François Bodart, Professeur

Septembre 1996

Production of Online Comments for C++ Reusable Components

Nicolas STROUBOULIS
Bachelor and Master of Computer Science Thesis.
Institut of Computer Science.
University of Namur.

Abstract

There is a common agreement that increased successful reuse will lead to higher quality software at lower cost. Classification is considered as an important technique of supporting the retrieval of software components for reuse. The main problem concerning finding software components for reuse, which are stored in a repository, like the Software Information Base (SIB) is how this repository is organised to support reuse. Structuring techniques used in object-oriented systems, such as instantiation, specialisation and attribution are helpful in tackling the problem. TELOS, the representation mechanism used for the SIB organisation provides such powerful means to structure collections of reusable components.

The purpose of this thesis is to design and implement a technique for reuse C++ reusable components comments. It's used within the EspritII ITHACA project for the description of software components that are stored in a repository, the software information base (SIB). A program, the *Parser* reads the C++ source code and extracts all different kind of comments that are related to the reusable components. Different keywords precede the comments in order to help the *Parser* to recognise the different kind of comments. These comments are then specified in SIB using Telos, the language for knowledge representation which provides efficient mechanisms to describe semantic

networks. SIB contains several predefined types classified according to the STA (Static Analyser) classification scheme. In SIB, our comments are defined as instances of these types. Then, the STA user interface is used to support the search and retrieval of the specified comments.

Supervisor : François Bodart, Professor.

Production de commentaires en ligne de composants réutilisables C++

Nicolas STROUBOULIS
Mémoire de licence et maîtrise en informatique.
Institut d'informatique.
Facultés Notre-Dame de la Paix, Namur.

Préface

Il a un accord commun de dire que la réutilisation va mener à des logiciels de meilleure qualité à moindre coûts. La classification est considérée comme une technique importante pour le support de la recherche de composants réutilisables. Le problème principal dans la recherche de composants logiciels réutilisables, souvent stockés dans des « repository » tel que SIB (Software Information Base) dépend de la manière dont ce « repository » est organisé pour supporter la réutilisation. Des techniques de structuration utilisées dans des systèmes orientés objets, telles que l'instantiation, la spécialisation et l'attribution sont très utiles pour cerner le problème. TELOS, le mécanisme de représentation utilisé pour l'organisation de SIB fournit de tels mécanismes puissants pour structurer des composants réutilisables.

L'objectif de ce mémoire est de concevoir et d'implémenter une technique pour réutiliser des commentaires de composants réutilisables C++. Ces techniques seront utilisées dans le projet EspritII ITHACA qui supporte la description de composants logiciels enregistrés dans le « repository » SIB, Software Information Base. Un programme, le *Parser* lit le code source C++ et extrait les différentes descriptions des composants classifiables.

Afin que le *Parser* reconnaisse les différents types de commentaires, on a précédé les commentaires réutilisables de différents mots clés. Ces commentaires sont alors spécifiés dans SIB en utilisant Telos, le langage de représentation de connaissances qui fournit des mécanismes efficaces pour la description de réseaux sémantiques. SIB contient plusieurs types prédéfinis classifiés d'après le schéma de classification STA (Static Analyser). Dans SIB, nos commentaires sont alors définis comme instance de ces types. Enfin, l'interface utilisateur de STA sera utilisée pour supporter la recherche de ces commentaires spécifiés.

Promoteur : François Bodart, Professeur.

Remerciements.

Je voudrais tout d'abord remercier mon promoteur M. François Bodart qui m'a donné l'occasion de réaliser ce mémoire ainsi que pour tous ses judicieux conseils et sa patience pendant toute la durée de ce travail.

Je voudrai également remercier M. Baudoin Lecharlier pour ses conseils lors des dernières modifications de ce mémoire.

Je tiens à remercier mes maîtres de stage de FORTH (Foundation of Research and Technology - Hellas), M. Panos Constantopoulos et M. Martin Doerr ainsi que toute l'équipe de recherche ITHACA pour leur fructueuse collaboration lors de mon stage en Crète. Je pense particulièrement, à Manos Etheodorakis, Polivios Klimathianakis et Yannis Tsitsikas pour leur soutien jusqu'au terme de ce document.

Je voudrai remercier William Poos pour son soutien et son amitié pendant toute la durée de mes études et évidemment ma famille pour leur patiente et le support qu'ils m'ont accordé.

Finallement, je tiens à remercier les Facultés Universitaires Notre-Dame de la Paix et l'Institut d'Informatique pour la qualité de l'enseignement que j'ai reçue.

Table des matières.

1. INTRODUCTION.

1.1. Généralités.

La classification est considérée comme une technique importante pour supporter la recherche de composants logiciels devant être réutilisés. Cependant, un des problèmes majeurs dans la recherche de tels composants, souvent stockés dans un répertoire ou dans une librairie, est la manière dont cette librairie est organisée. Les critères de classification que nous allons considérer ici se basent sur les fonctionnalités des composants c'est-à-dire sur le service rendu par ces composants. Les techniques de structuration utilisées dans les systèmes orientés objets, telles que l'instantiation, la spécialisation ou l'attribution sont très utiles pour cerner ce problème.

Néanmoins, ces techniques de structuration, ainsi que la nature des objets eux-mêmes n'aident pas pleinement l'utilisateur dans sa tâche de *sélection,* une des étapes de base dans la réutilisation. Ainsi, malgré la grande assistance que peut apporter le processus de cascade (browsing) en fournissant plusieurs possibilités de liaisons entre les objets, il n'apporte qu'une aide limitée à la recherche. Les utilisateurs doivent identifier eux-mêmes les composants adaptés à leurs besoins.

Puisque les besoins des utilisateurs concernent généralement la fonctionnalité de l'information réutilisée, souvent exprimée en mots courants, nous allons analyser ici un système qui supporte la recherche de types documentaires d'objets réutilisables.

1.2. Commentaires en ligne.

L'analyse qui va suivre représente mon travail de fin d'études de licence et maîtrise en informatique. Il fait suite à un stage de quatre mois réalisé à ICS-Forth (Institute of Computer Science - Foundation Of Research and Technology - Hellas), à l'Université de Crète (Grèce).

J'étais intégré dans une équipe de chercheurs qui étaient impliqués dans le développement d'un outil générique : SIS (Semantic Index System) développé dans le cadre du projet Esprit ITHACA[1] pour supporter différents types de description de logiciels. Parmi les composants de l'environnement jouant un rôle important dans cette analyse, on trouve les éléments suivants :

- **SIB** (System Information Base) : SIB a été spécialement conçue pour supporter le développement de différents systèmes du projet ITHACA. SIB peut être considéré comme un système d'information à fonctionnalités complexes (un peu comme une grande base de données) qui facilite le stockage, le partage, l'accès, la gestion et le contrôle des données. Il peut aussi bien représenter des spécifications de composants logiciels avec leurs liens, des composants logiciels avec leurs liens ainsi que les liens entre les composants et leurs spécifications. Une autre caractéristique est sa capacité de représenter des composants multimédias.

[1] Integrated tool for highly advanced computer applications.

- **STA** (Static Analyser) : outil qui supporte l'analyse statique du code représenté dans SIB. STA définit un modèle de classification de composants basé sur des facettes. Une facette représente un critère de classification et est défini par un ensemble de termes qui ont des relations hiérarchiques (généralisation /spécialisation). Le résultat de l'analyse statique est représenté sous forme textuelle ou graphique (browsing) permettant ainsi de rechercher et de naviguer entre les différents composants.

- **Telos** : le langage de représentation des connaissances utilisé pour la spécification des composants de SIB. Il est utilisé pour décrire et organiser de manière uniforme des composants logiciels. Il est basé sur un modèle orienté objet et fournit trois principes de classification de composants dénommés classification (instantiation inverse), spécialisation (généralisation inverse) et l'agrégation (décomposition inverse). Il ne fait pas de distinction entre le schéma et les données. On peut ainsi modifier le schéma à n'importe quel moment par de simples requêtes sans perte de données.

En se basant sur la classification STA des composants SIB, l'objet de ce travail est de construire un automate qui pourra dégager des commentaires les propriétés à affecter à la classification. Ces commentaires peuvent être réutilisés à d'autres fins, notamment pour obtenir une documentation en ligne des composants réutilisables.

Le schéma ci-dessous illustre bien les différentes étapes nécessaires à l'obtention de commentaires en ligne. Ainsi, considérons un programme C++ dans lequel nous avons définis différentes classes ou méthodes qui nous semblent utiles pour la classification. La première étape consiste à utliliser la méthode de classification STA (voir 3.2 et 3.3) pour classifier les différents composants de ce programme. Le résultat de cette classification résulte en un ensemble de composants classifiés. Chaque composant logiciel du programme ou librairie est représenté dans le schéma de classification STA par un nom logique.

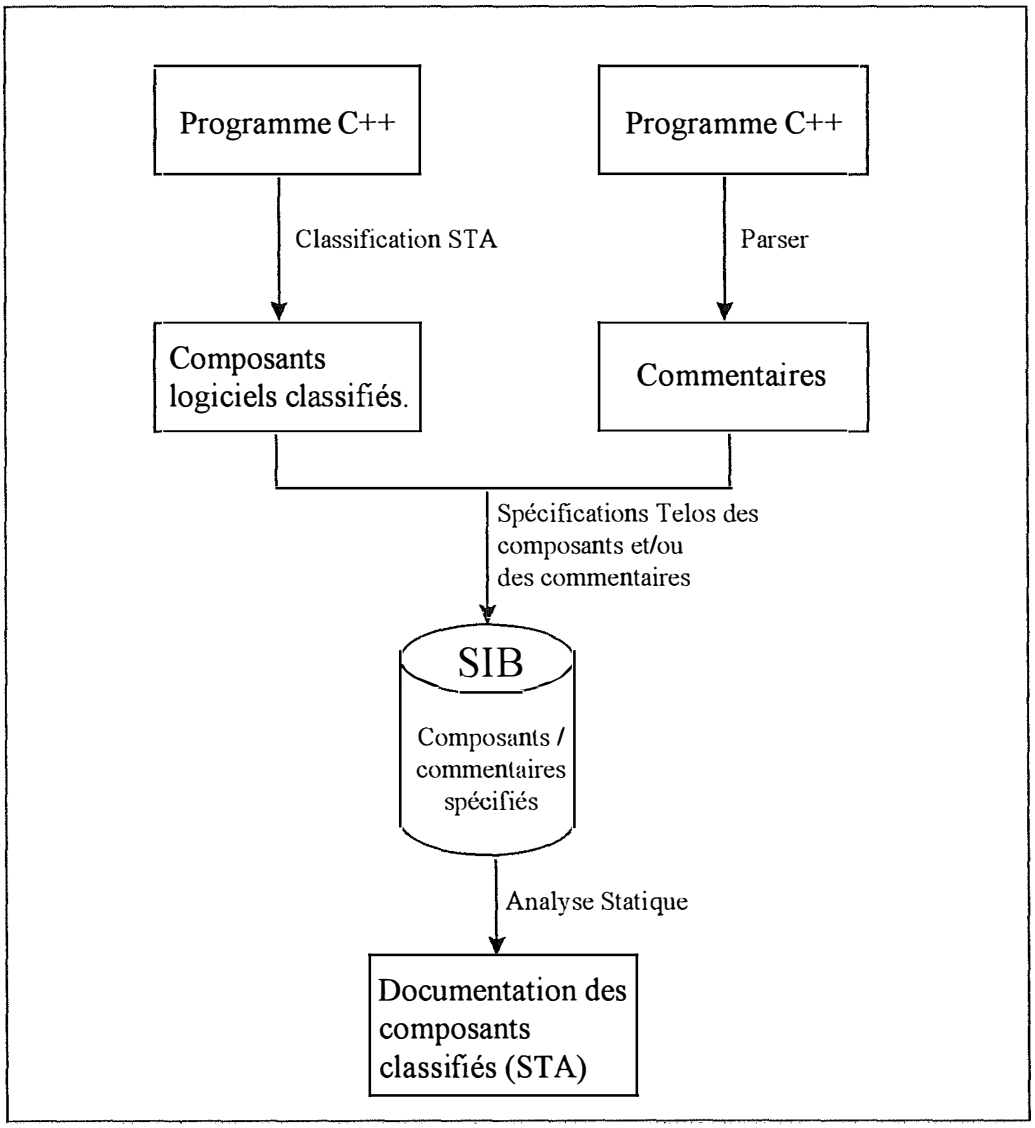

Figure 1. : Schéma général des composants/ commentaires en ligne.

L'étape suivante consiste alors à spécifier ces composants à l'aide du langage de représentation de connaissances Telos (voir 3.5). Bien que le projet Esprit ITHACA se base sur la description de logiciels écrits en C++, Cobol ou Cool[2], le présent document ne s'intéressera qu'aux commentaires de logiciels écrits en C++. Cette spécification nous

[2] L'environnement COOL est un environnement de développement de logiciels spécialement adapté pour le développement de systèmes très complexes. Pour la conception et le développement, il est principallement basé sur une approche orientée objet.

permet donc d'obtenir des composants dans SIB. L'analyse statique effectuée par le logiciel Static Analyser nous permet enfin d'obtenir des composants en ligne.

Concernant les commentaires, l'approche est un peu différente. A partir du programme C++, on va utiliser un automate (le *Parser*) qui viendra lire le code source et en dégagera les commentaires classifiables. Toute la logique concernant les relations qui existent entre ces différents commentaires devra être traitée par ce programme. Il s'agira ensuite à utiliser Telos pour spécifier ces commentaires dans SIB. Le logiciel STA n'aura alors qu'à afficher ces commentaires en ligne en utilisant les informations stockées dans SIB.

1.3. Organisation du travail.

Le chapitre 2 va présenter l'apport des développements orientés objets pour la réutilisation.

Le chapitre 3 présentera le système SIS (Semantic Index System). Il décrira brièvement l'interface graphique de SIS (le logiciel " Static analyser (STA)") ainsi que son modèle de classification basé sur des critères sémantiques et adaptables aux composants orientés objets. Il décrira ensuite un exemple de classification qui respecte ce modèle. Ce chapitre se terminera par la présentation générale du langage de représentation de connaissances Telos et de la spécification du langage C++ en Telos.

Le chapitre 4, sur base de la classification décrite au chapitre précédent, présentera les caractéristiques de la documentation en ligne ainsi que la méthode utilisée pour obtenir des commentaires en ligne. On y présentera en outre les spécifications et la grammaire détaillée de l'automate décrit précédemment, le *Parser*.

L'annexe 1 présentera le programme source du *Parser* écris en C.

Enfin, l'annexe 2 comporte un exemple de programme C++ où les commentaires réutilisables peuvent être traités par ce *Parser*.

2. CLASSIFICATION DE COMPOSANTS ORIENTES OBJETS

2.1. Introduction

Il y a un accord commun [KARL92] de dire que la réutilisation va mener à des logiciels de meilleures qualités à prix réduits. Les techniques de développement orientés objets apportent divers avantages telles que les techniques d'héritage, d'encapsulation, ou la limitation du fossé sémantique qui existe entre la conception et l'implémentation. Ainsi, dans un contexte orienté objet, la réutilisation nécessite la classification de composants afin qu'ils puissent être retrouvés lors de la réutilisation.

Par réutilisation, on signifie que quelque part, dans le processus de développement d'un logiciel, des produits issus de développements antérieurs sont réutilisés. Tout en ne se souciant pas encore de ce qui sera réutilisé et à quel moment, on peut déjà affirmer que la réutilisation suppose :

- augmenter la productivité
- améliorer la qualité.

En effet, si le développement d'un produit réutilisé a nécessité un certain travail, alors ce travail est quasi épargné chaque fois que l'on réutilise ce produit. De même, on peut améliorer la qualité puisque le même produit sera utilisé et testé dans différents contextes.

Il faut cependant tenir compte du surcoût engendré par le développement d'objets réutilisables : surcoût de conception et de développement.

2.2. Le processus de réutilisation.

2.2.1. Différentes approches de réutilisation.

Si on considère ce qui doit être réutilisé et à quelle phase de développement cela doit être réutilisé, on peut identifier deux approches principales de réutilisation. La première consiste à utiliser des ***générateurs d'application*** [KARL92]. Cette approche cache cependant le processus de réutilisation pour l'utilisateur. La caractéristique de cette approche est que des connaissances approfondies dans un certain domaine sont codées en un langage spécifique pour ce domaine. En d'autres mots, on peut considérer l'utilisation d'un générateur d'application comme une programmation à un haut niveau d'abstraction. Ainsi, différents langages de quatrième génération comme LEX[3] ou YACC[4] sont de bons exemples de générateurs d'application. Cependant, l'inconvénient de cette approche est qu'un générateur est spécifique pour un certain domaine et que le développement coûte très cher et requiert une connaissance très approfondie du domaine traité.

[3] Lex (Lexical analyser) : Il génère un programme C pour des analyses syntaxiques simples. (voir manuel utilisateur UNIX).

[4] Yacc (Yet Another Compiler-Compiler) : il est utilisé pour la création de " PARSER ". En lisant des spécifications grammaticales, il génère un programme C (voir manuel utilisateur UNIX).

La seconde approche est la ***réutilisation par composition*** [KARL92] qui signifie que l'utilisateur prend une participation active dans le processus de classification. Ici, l'utilisateur décide ce qui doit être réutilisé et quand cela doit l'être. La réutilisation par composition signifie que l'utilisateur sélectionne des composants et les utilise lors du développement d'une nouvelle application. Ce modèle fournit donc une approche plus générale qui dépend moins du domaine traité. De même, il n'y a aucune limitation sur ce qui peut être réutilisé et à quelle phase de développement cela peut être réutilisé. De cette manière, la sélection des composants incombe à la responsabilité de l'utilisateur. Ainsi, afin que l'utilisateur puisse les retrouver le plus rapidement possible, certaines contraintes doivent être respectées:

- les composants doivent être bien définis, bien documentés et doivent avoir une interface simple et claire.

Comme on le verra plus en détails dans les chapitres suivants, ces aspects sont très importants puisque l'utilisateur doit comprendre, évaluer et introduire ces composants dans un nouveau produit sans trop d'effort.

2.2.2. Développement orienté objet et réutilisation.

Le modèle de *réutilisation par composition* a été d'autant plus utilisé une fois que la philosophie orienté objet a été acceptée à grande échelle. Une des raisons majeures est que le développement orienté objet crée des *objets* qui ont les mêmes propriétés que celles que nous venons d'exposer pour les composants réutilisables. Il y a donc une correspondance naturelle entre les composants produits et les composants réutilisables. Cette correspondance n'est pas si bien définie quand nous considérons des langages de programmations traditionnels où les composants réutilisables sont des modules, procédures, fonctions, etc. Dans le contexte orienté objet, les objets sont définis sous

forme de classes permettant ainsi aux composants réutilisables d'avoir une interface uniforme.

Une autre caractéristique importante du développement orienté objet est la relation étroite et le faible écart sémantique qui existe entre l'analyse/conception et l'implémentation. Cela permet de réutiliser des composants beaucoup plus tôt dans les phases de développements informatiques. Ce n'est par contre pas le cas si nous considérons des techniques plus traditionnelles comme les diagrammes de flux ou la décomposition en fonction où la différence entre la conception et l'implémentation ne permet pas d'identifier des composants réutilisables aussi tôt dans la phase d'analyse/conception.

L'avantage à utiliser un composant dans une phase précoce est que si nous pouvons par exemple réutiliser le *design*[5] d'un objet, nous pourrons également réutiliser l'implémentation correspondante puisque l'implémentation est l'extension logique de la conception. Ainsi, plus nous pouvons réutiliser du travail et plus nous épargnerons du temps. Si cependant, il n'y a pas d'objet existant qui nous convienne, le développement orienté objet nous permet de le voir dès la phase conceptuelle. Ceci serait difficile avec les techniques de développement traditionnelles.

De même, la proximité qui existe entre l'analyse orienté objet et l'implémentation peut également supporter certains changements au modèle conceptuel sans que l'implémentation ne devienne obsolète. Cela est dû au fait qu'un changement dans le modèle conceptuel ne se propage pas dans une grande partie du programme. Si les changements sont localisés dans un objet conceptuel, on pourra facilement les localiser à l'implémentation. Une approche traditionnelle peut nécessiter beaucoup plus de changements à l'implémentation même pour des modifications mineures à la conception.

[5] Le modèle conceptuel

Comme nous l'avons mentionné, les objets crées par des développements orientés objets ont des propriétés propices pour la réutilisation. Il est également important de savoir que d'autres produits appartenant au processus de développement, comme par exemple l'analyse ou les schémas conceptuels, peuvent également être réutilisés. En particulier, les chapitres suivants vont proposer une manière de réutiliser les commentaires des composants réutilisables. Or, réutiliser de tels composants signifie donc réutiliser des classes. Cela permet également d'exploiter le mécanisme d'héritage pour créer une sous-classe à partir d'une classe existante. Cette sous-classe hérité les propriétés de la classe parent mais peut être changée afin de répondre aux besoins spécifiques des utilisateurs. Nous pouvons ainsi identifier trois approches de réutilisation de classe dans le modèle de *réutilisation par composition.*

- réutilisation telle quelle c'est-à-dire réutilisation de la classe sans modification.
- réutilisation par sous-classement.
- réutilisation par modification c'est-à-dire en modifiant la fonctionnalité du composant.

La première possibilité consiste à réutiliser la classe directement sans la modifier. C'est évidemment la solution la plus profitable mais malheureusement, les classes existantes ne répondent pas toujours exactement aux besoins des utilisateurs. Cela nous mène donc à la deuxième approche où une classe est réutilisée par sous-classement. Nous pouvons ainsi ajouter ou modifier certaines fonctionnalités afin d'adapter les propriétés de la classe parent à nos besoins. La troisième solution est la moins profitable puisqu'il s'agit de modifier le code source de la classe réutilisée.

Nous avons ainsi identifié les critères qu'un composant doit remplir afin qu'il soit utile pour la réutilisation. Seulement, la nature des composants eux-mêmes ne supporte pas l'utilisateur dans sa tâche de sélection de composants réutilisables. Nous avons donc besoin d'un certain support qui viendrait aider l'utilisateur dans cette tâche de recherche et de sélection.

2.2.3. La réutilisation implique la classification.

Habituellement, les composants réutilisables sont situés dans une librairie ou un "repository" comme SIB (Software Information Base) [Cons92] et l'utilisateur recherche dans cette librairie les composants qui correspondent au mieux à ses besoins. Or, le problème principal concernant la réutilisation de tels composants, c'est la manière dont cette librairie sera organisée afin qu'elle supporte la réutilisation. Si on considère la réutilisation de classes, on sait qu'elles ont un certain ordre qu'on appelle *hiérarchie de classes*. TELOS (section 3.4.), le mécanisme de représentation utilisé pour l'organisation de SIB fournit de tels moyens performants qui servent à structurer une collection de composants réutilisables. Ainsi, afin de supporter la recherche de composants réutilisables, on a besoin d'un *schéma de classification*. Par schéma de classification, on considère un ordre de composants où les composants ayant une signification proche sont regroupés et caractérisés par des propriétés communes. Cela permet à l'utilisateur d'entrer les propriétés que l'objet réutilisable devrait avoir en vue d'obtenir une liste de composants potentiels. L'utilisateur recherche donc certaines propriétés plutôt que certains objets.

2.2.4. Schémas de classification.

Puisqu'il n'y a pas de classification intrinsèque des composants logiciels, on doit alors " inventer " un ordre basé sur certains critères. Habituellement, il s'agit de choisir un ou plusieurs aspects ou facettes d'un composant dans lesquels existe une certaine structure ou dans lesquels une structure peut être définie. Il y a plusieurs manières de procéder. Dans le cas simple, c'est quand on ne choisit qu'un aspect ou facette. Le domaine qui doit être classifié est alors décomposé en plusieurs catégories qui peuvent à leurs tours être re-décomposées, etc. Cela nous donne une classification facile à comprendre et à utiliser et qui ne nécessite pas nécessairement le support de l'informatique. Ce type de schéma est dénommé *schéma énumératif*.

Par exemple, on peut considérer la classification des fichiers dans un répertoire. Un disque contient plusieurs répertoires dont chacun peut également contenir d'autres répertoires. On obtient ainsi un schéma de classification simple à la manière du gestionnaire de fichiers de Windows.

Par contre, les *schémas basés sur des facettes* représentent un autre type de classification. La différence principale avec les *schémas énumératifs* est qu'un schéma basé sur des facettes nous permet de voir un composant sous différents points de vue ou facettes. Chaque facette dénote une certaine propriété du composant. La valeur d'une facette est appelée *terme*. Les termes possibles pour une facette peuvent être reliés ensemble de telle manière qu'ils dévoilent les relations qui existent entre les différents composants; composants qui sont classifiés avec des termes respectifs. Ainsi, un modèle basé sur des facettes change en fonction des composants qui sont classifiés. Par contre, les schémas énumératifs sont plus stables parce que l'introduction de nouvelles catégories peut causer des problèmes.

Considérons les relations suivantes :

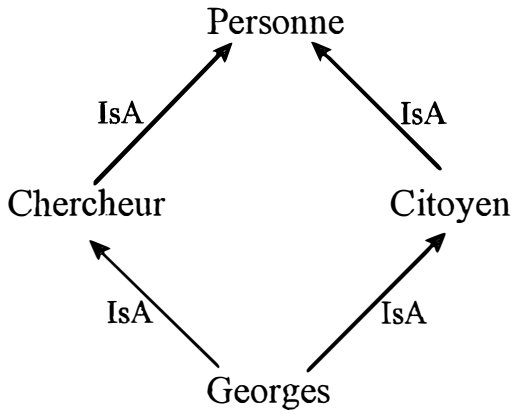

Figure 1. : Schéma basé sur des facettes.

Nous pouvons définir des relations hiérarchiques décrivant un *Chercheur* ou un *Citoyen* comme étant tout deux des *Personnes.* De même, pour l'instance *Georges*, on a défini deux catégories ou facettes. Il est soit repris dans le groupe des *Chercheurs*, soit dans le groupe des *Citoyens*.

De même, il faut bien être conscient que, pour la recherche de composants, cette technique requiert un support informatique. La nature dynamique d'un schéma basé sur des facettes n'est pas facilement gérable sans support informatique puisqu'il peut y avoir plusieurs facettes et des relations complexes entre les termes.

3. LE MODELE DE CLASSIFICATION DU SIS "Semantic Index System".

3.1. Environnement.

Le SIS (Semantic Index System) est un outil générique développé par FORTH, Crète dans le cadre du projet Esprit ITHACA[6] pour supporter différents types de **descriptions de logiciels**. C'est un système orienté objet hautement optimisé pour mémoriser et naviguer à travers des structures transitives en forme d'arbre tels que les

[6] ITHACA (Integrated Toolkit for Highly Advanced Computer Applications) est un projet d'intégration technologique qui a débuté en 1989 et fondé par la Commission Européenne comme faisant partie du programme ESPRIT II. Le consortium réunit des leaders de la vente de technologies de l'information (IT), des maisons de logiciels spécialisées ainsi que des institutions de recherche. Parmi les partenaires, on y trouvent Siemens Nixdorf Informationssysteme AG (Allemagne), Bull S.A. (France), Datamont S.p.A. (Italie), TAO S.A. (Espagne), F.O.R.T.H. (Grèce) et l'Université de Genève, Centre Universitaire d'Informatique (Suisse).

arbres d'appel de procédure, les hiérarchies de classes, les structures de référence d'un schéma de base de données, etc. Il permet la libre déclaration d'un schéma (metaclasse) au dessus du schéma primaire, supportant ainsi la déclaration de caractéristiques communes ou distinctes d'une partie du schéma primaire. Ses intéressantes performances sont essentiellement dues à l'utilisation de mémoires caches et de liens bidirectionnels. Il tente actuellement de spécialiser ses applications en tâches spécifiques afin d'intéresser des exploitations industrielles.

En considérant que la quantité d'information traitée par et entre les organisations n'arrête pas de croître considérablement, les organisations impliquées dans ITHACA ont pensé que les méthodes traditionnelles ne pourront pas répondre à la complexité et aux problèmes que ce flux d'information va causer. Ils ont pensé qu'une nouvelle approche était cruciale à moyen terme. En conséquence, ils ont développé un système de support aux applications basé sur des technologies avancées dans le domaine de la programmation orientée objet et concernant plus particulièrement les langages de programmation, les technologies de base de données, les systèmes d'interface utilisateurs et les outils de développement de logiciels. ITHACA inclut un ensemble d'outils intégrés qui exploitent les avantages des technologies orientées objets pour promouvoir la réutilisation et "l'intégrabilité", facteurs qui s'annoncent très importants pour l'amélioration de la productivité et de la qualité des logiciels.

3.1.1. Composants de l'environnement.

Parmi les composants impliqués dans SIS, on y trouve SIB (Software Information Base), STA (Static Analyser) et un langage de représentation des connaissances : TELOS.

- SIB (Software Information Base) : c'est une base de données contenant toutes sortes de composants logiciels. Elle a été conçue de telle manière à faciliter le développement d'applications basées sur la réutilisation. De telles bases de

données peuvent être considérées comme des systèmes d'information à but précis, utilisés pour le support de modèles sémantiques puissants et pour une recherche flexible de composants [CDV93].

- STA (Static Analyser) : le logiciel Static Analyser est un outil de SIS pour la documentation et l'analyse de propriétés statiques de codes sources écrits dans différents langages. Il a été utilisé pour traiter du code écrit en COBOL, C++ et Cool. Le logiciel Static Analyser est considéré comme une spécialisation de SIB. Dans notre cas, il sera utilisé pour l'analyse statique des commentaires écrits en C++ permettant d'obtenir des commentaires en ligne (section 3.2).
- TELOS : le langage de représentation des connaissances utilisé pour la spécification des composants de SIB. Il est utilisé pour décrire et organiser de manière uniforme des composants logiciels (section 3.4).

Notons également que dans un environnement de développement de logiciel intégré, l'ensemble minimal d'outils de SIS consiste en :

- un éditeur guidant la syntaxe (a syntax driven editor : SDE).
- un ensemble de compilation (compilateur, assembleur, éditeur de liens).
- un " debugger " symbolique.
- un analyseur statique (le Static Analyser : STA).

3.2. L'analyseur statique de SIB (STA).

3.2.1. L'interface d'analyse graphique de SIS .

L'analyseur statique de SIB (STA : pour STatic Analyser) est un des outils de SIS qui supporte l'analyse statique du code représenté dans SIB (Software Information Base) [CONS92].

A l'exception de l'entrée des données, des interfaces de requêtes programmées et interactives, le SIS dispose d'un système d'affichage graphique. Il présente des structures transitives relatives à des critères arbitraires définis par l'utilisateur. Par exemple, on peut avoir un arbre des appels affichant les noms des fichiers où les procédures sont définies.

Cet interface utilisateur coopère avec le processeur de requêtes[7] de SIS. Les informations de SIS peuvent être retrouvées en exécutant une des requêtes prédéfinies disponible dans un menu de l'interface utilisateur. Quand une requête est exécutée, le processeur de requêtes extrait les données de SIB et affiche le résultat sur l'écran. Ce résultat peut apparaître sous deux aspects : en mode textuel sur une fenêtre Motif standard ou en mode graphique. L'interface graphique que dispose SIS, le "Static Analyser" présente des structures transitives se basant sur des critères définis de manière arbitraire par les utilisateurs. Par exemple, un arbre des appels de fonctions serait composé des noms des fichiers contenant ces procédures. Toutes les informations sont présentées à l'utilisateur dans les termes qu'il a l'habitude d'utiliser dans son environnement spécifique.

[7]Il exécute des requêtes prédéfinies dans le menu ou alors il exécute des requêtes basées sur des critères définis par l'utilisateur, à l'aide de l'interface utilisateur en remplissant la zone " Target query " (voir figure 3.1).

De plus, une requête peut avoir un paramètre sur lequel elle opère. Ce paramètre apparaît sur la figure suivante comme "Query Target" et doit être un objet existant dans SIB. Ce paramètre est toujours visible et peut à tout moment être modifié par l'utilisateur.

Ainsi, nous voyons que le composant "CoolExternalVariable" (variable externe Cool) est représenté à la figure suivante comme noeud central. Au dessus figurent les objets auxquels il fait référence et en dessous, tous les noeuds se référant à celui-ci. Notons également que le lien affiché en gras représente une relation isA c'est-à-dire qu'une *Variable Cool Externe* **est une** *Variable Cool.*

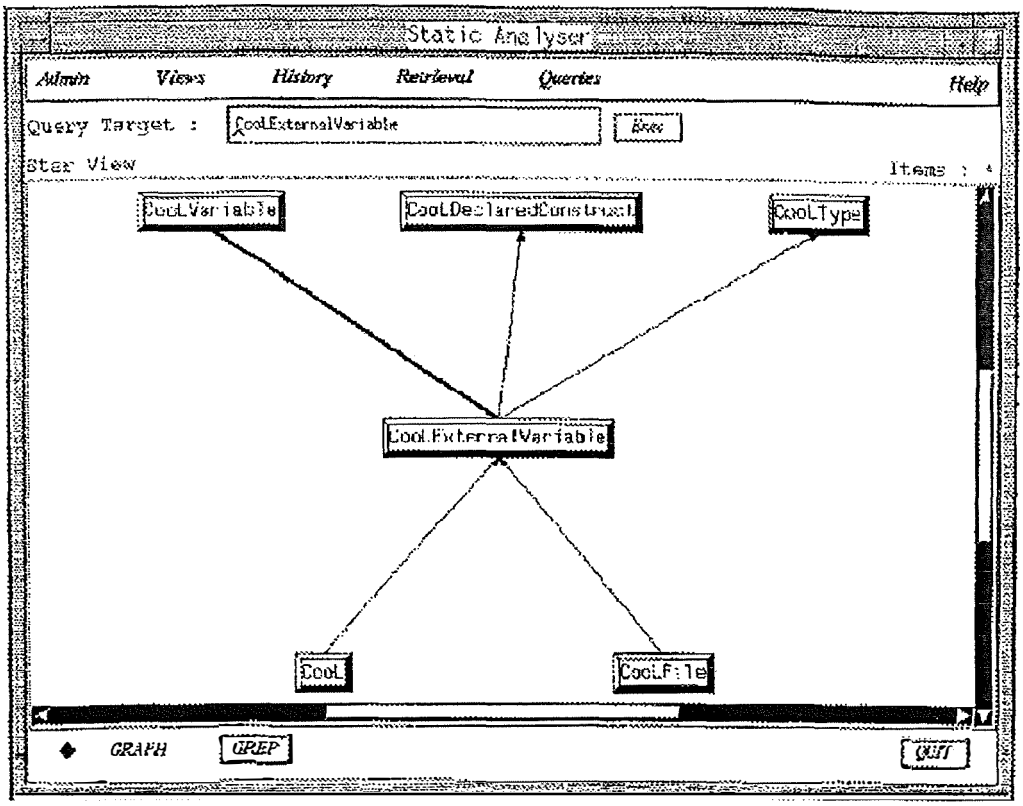

Figure 3.1 : Affichage graphique sous forme d'étoile de l'objet "CoolExternalVariable" [DKP94].

3.2.2. Le schéma de classification STA.

Généralement, afin de faciliter la recherche et les requêtes, on a proposé différentes approches de classification de composants logiciels. Cependant, puisque les besoins des utilisateurs concernent les sémantiques des informations à réutiliser, souvent exprimées en mots courants, nous avons également besoin d'un *système de traitement de la sémantique* qui supporte la recherche et la réutilisation des objets réutilisables.

En ce qui concerne le modèle de classification de STA, il respecte le schéma de classification[8] basé sur des facettes. Les composants sont décrits par des **facettes** de base. Comme on a vu précédemment, une facette représente un critère de classification et est définie par un ensemble de **termes** formant entre eux des liens hiérarchiques de généralisation ou de spécialisation.

3.2.2.1. Les facettes et les espaces de termes structurés.

Les facettes forment les points d'entrées du schéma de classification STA. Une **facette** représente souvent une perspective, un point de vue ou une dimension d'un domaine particulier. Dans le schéma de classification STA, une facette concerne certains aspects ou propriétés des composants de SIB. Une facette est définie comme un ensemble fini de valeurs qui sont en relation appelés **termes**. L'ensemble des facettes utilisées pour classer les objets de SIB définissent un **espace de facettes**. Chaque

[8] *Classification fonctionnelle* (taxonomie ou catalogage) est une organisation systématique de concepts représentés sous forme de mots clés où les informations ayant les mêmes sémantiques sont regroupées ensemble.

Les schémas de classification fournissent des techniques pour l'ordonnencement systématique.

composant logiciel de SIB est caractérisé par un ensemble fini de paires (f,t), où f est une facette dans un espace de facettes données et t est un terme de f. Pour éviter une classification ambiguë, on assigne un seul terme par composant classifiable. L'ensemble des termes structurés qui est associé à chaque facette défini un **espace de termes.** Les termes qui appartiennent à n'importe quel espace de termes d'une facette peuvent être utilisés pour la classification ou pour la recherche.

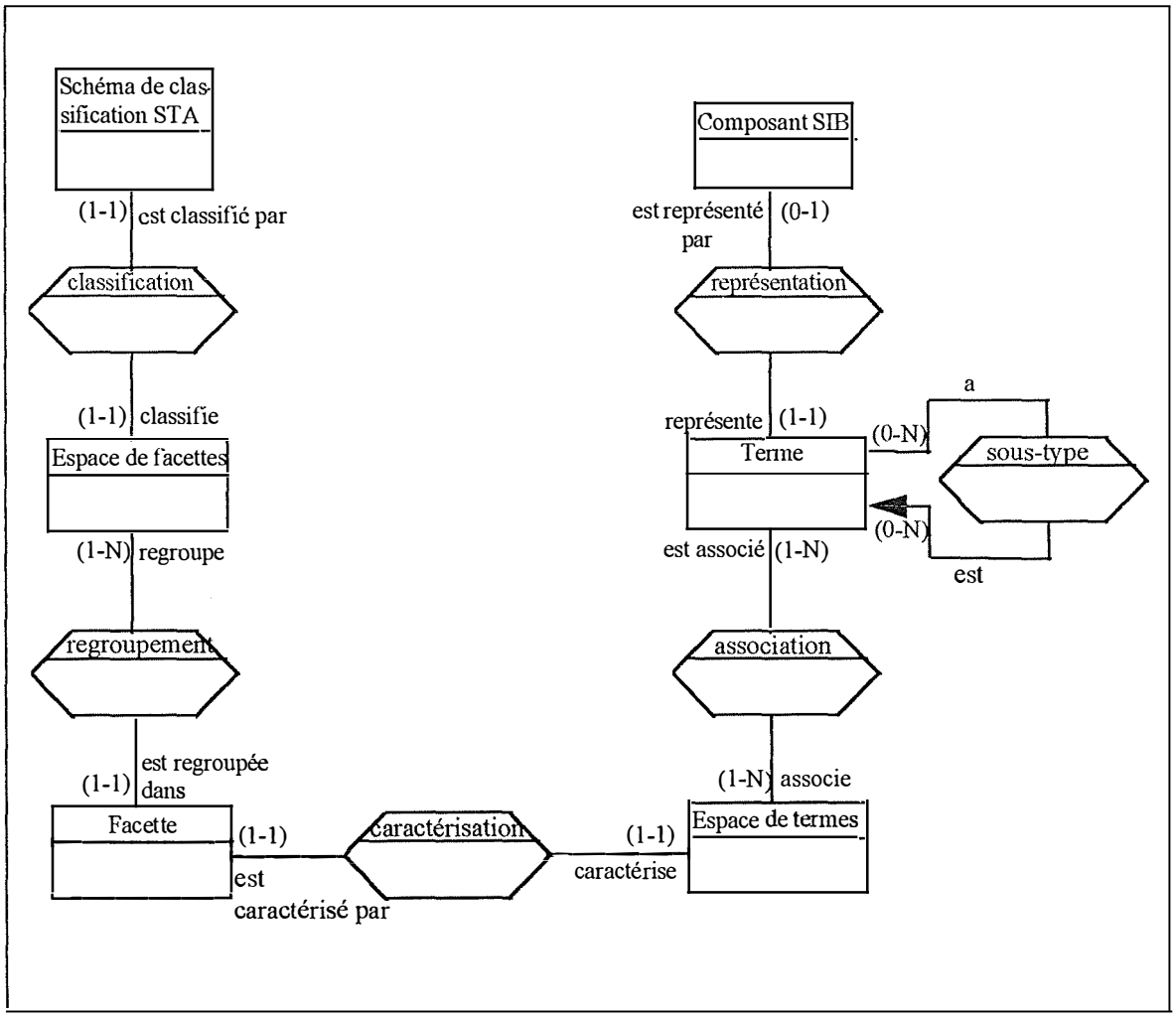

Figure 3.2 : Schéma Entité - Relation de la classification STA.

Chaque espace de termes d'une facette est organisé d'après la hiérarchie généralisation/spécialisation incluant plusieurs héritages dans le but de respecter les points suivants :

• un espace de termes doit être applicable globalement dans un même domaine d'application, couvrant aussi bien les concepts généraux que spécifiques.

• les concepts doivent être organisés de manière à respecter des caractéristiques communes ou des comportements identiques.

• le schéma de classification doit être simple et bien défini tout en étant aussi complet que possible.

Dans le schéma de classification STA, les termes ayant des caractéristiques ou des comportements communs sont considérés comme sous-classes d'une classe parent commune (terme). Respectivement, une classe peut être un membre d'autres groupes ayant des caractéristiques communes plus génériques, entre plusieurs classes (termes) plus génériques, etc. La spécification d'un terme feuille hérite automatiquement des termes plus abstraits. Pour un terme particulier, ceci permet de minimiser les entrées de données utiles.

Cette hiérarchie isA (est un) fournit une aide sur la sémantique de chaque terme ainsi qu'un système de recherche des termes eux-mêmes. Ceci ne défend pas une recherche "plate" des termes c'est-à-dire une recherche de mots par ordre alphabétique à la manière des mots clés. Cette hiérarchie peut être organisée sous une forme d'arbre c'est-à-dire une structure simple avec des relations de généralisation/spécialisation. Une structure plus générale forme un graphe acyclique, où un terme peut être généralisé par plus d'un autre terme.

Ci-dessous, nous illustrons un exemple où un terme est généralisé par plus d'un autre terme [DOER93].

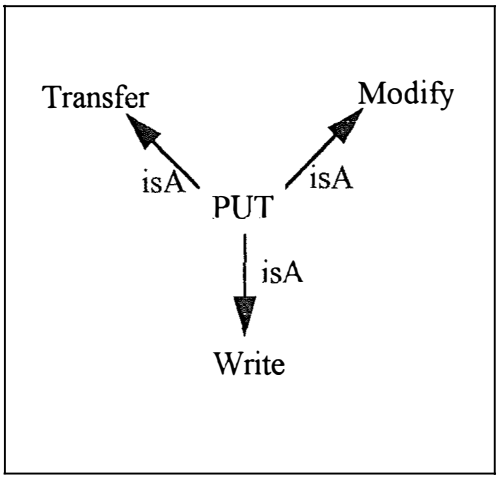

Figure 3.3 : Une partie d'un graphe acyclique - un espace de termes structurés.

3.2.2.2. Les facettes de STA.

Comme on l'a vu précédemment, la classification est nécessaire pour ordonner les composants qui devront être réutilisés. Seulement, puisqu'il n'y a pas d'ordre intrinsèque pour les composants logiciels, nous devons inventer un ordre basé sur certains critères. La sélection de certaines facettes est bien souvent arbitraire puisqu'elle ne donne qu'un ordre artificiel des composants. Ainsi, on aurait pu choisir comme facette le nombre de lignes et l'origine géographique, mais cela n'aurait pas été très approprié pour la recherche de ces composants. Lors de la réutilisation, le but de la classification est de sélectionner un petit ensemble de composants. Cet ensemble devra contenir des composants candidats puisqu'il ne s'agit pas de trouver l'élément final en une fois. Cependant, il est très important de ne pas oublier de composants potentiels.

En ce qui concerne la solution choisie pour STA, son schéma de classification utilise les facettes définies dans le projet REBOOT (REuse Based on Object-Oriented

Techniques). La classification de REBOOT est basée sur des facettes similaires à celles définies par Prieto-Diaz [PDF87]. Prieto-Diaz s'intéressait surtout aux fonctionnalités des composants et basait sa classification sur le quintuple : fonction, objets, agent, type de système et domaine fonctionnel. Les trois dernières facettes fournissent des connaissances concernant l'environnement interne (où les composants des logiciels sont exécutés) et l'environnement externe (où les composants des logiciels sont appliqués). Ceci n'était pas directement applicable aux composants orientés objets de telle manière que REBOOT a défini d'autres facettes.

Le schéma de classification de STA est construit autour des quatre facettes proposées par REBOOT:

- Abstraction
- Operation
- OperatesOn
- Dependencies

Les trois premières facettes représentent les relations bien connues *sujet-verbe-complément.*

• Abstraction

Un composant orienté objet est bien plus considéré comme une chose, un substantif que comme une fonction ou un verbe. Ainsi, la meilleure façon de décrire/classer un composant serait de trouver un nom qui serait un objet, une chose du monde réel ou une abstraction de cet objet et qui s'adapterait au mieux à ce composant. C'est ce que nous définirons comme facette **abstraction**.

Exemples [KARL92]:

Repository

- Pile
- File
- Base de donnée
- Relationnel
- Orienté-objet

Systèmes Windows

- Macintosh
- MS-Windows
- X-Windows

- **Operation**.

Un composant orienté-objet offre généralement plusieurs méthodes qui caractérisent la fonctionnalité de ce composant. Une facette **Operation** contient une liste d'opérations offertes par un composant. Ceci est similaire aux **fonctions** de Prieto-Diaz mais puisqu'une classe possède plusieurs méthodes, il est naturel d'utiliser plusieurs termes pour la facette **Operation**. En général, l'espace des termes d'une facette **Operation** introduit des mots bien connus dans le domaine de l'ingénierie logicielle et décrit ce que fait une méthode (le *Quoi*).

Exemple : pour le composant " File ", on pourrait avoir comme facette **Operation** les termes suivants :

- Open
- Update
  - delete
  - save

- **OperatesOn**

La facette **OperatesOn** contient les objets avec lesquels une classe est en relation. Dans les programmes, ces objets peuvent être représentés comme argument en entrée et en sortie d'une fonction.

De même, puisque ce concept inclut entre autres des classes que l'objet va manipuler et qui sont elles-mêmes reprises dans une facette **Abstraction,** alors cette facette **Abstraction** constitue en quelque sorte un sous-ensemble de la facette **OperatesOn**.

- **Dependencies**

La facette **dependencies** va regrouper toutes les dépendances spécifiques qui vont limiter l'utilisation des composants. Cette facette correspond à la troisième facette de Prieto-Diaz. Pour STA, ces dépendances peuvent provenir du matériel, du système d'exploitation ou d'un langage de programmation. Il est évident qu'on aura quelques doutes sur la façon de remplir cette facette et l'espace de termes sera probablement moins bien structuré que précédemment (plus de hiérarchies mais plus petites). Voici quelques exemples de termes [KARL92]:

Langage de programmation

Lisp

Scheme

CommonLisp

Object-Oriented

C++

Smalltalk

Simula

Beta

Hardware

Sequential machines

Sun

Sun3

Sun4

3.2.2.3. Exemple simple de classification STA.

Lorsque l'utilisateur recherche un objet à réutiliser (en C++, on parlera de classe), il insère des termes pour les facettes qui l'intéressent. Imaginons donc le scénario suivant :

Soit un développeur qui veut construire un système de gestion des finances personnelles. Une partie du système est un objet qui tient la trace de toutes les factures qui ne sont pas payées. Le système devrait prévenir l'utilisateur lorsqu'une facture n'est pas payée et lorsque la facture est payée, le système devrait mettre à jour le montant restant en compte.

Ainsi, nous pouvons appeler la classe *transaction_handler* qui contient les méthodes[9] suivantes : *get_invoice* et *received_paid_invoice* qui agissent sur l'objet *invoice*. L'objet *transaction_handler* doit également coopérer (operate on) avec l'objet

[9] En C++, la méthode d'une classe est une fonction définie pour cette classe.

account qui peut recevoir le message *update_customer_account*. Il faut également fournir la méthode *notify_pay_invoice*.

De plus, les classes devraient être écrites en C++ et la conception des classes doit être conforme à la méthode de Grady-Booch.

Avec de telles spécifications, voici les objets que le développeur a besoin :

Abstraction	Operations	OperatesOn	Dependency
personnal finance transaction_handler	get_invoice notify_pay_invoice receive_paid_invoice update_customer_account	invoice account	C++ Brady-Booch

Evidemment, ces spécifications sont valables si les objets requis sont disponibles dans la librairie.

3.2.2.4. L'espace de termes de STA.

La sélection des termes doit être conforme aux règles suivantes :

- Les termes sélectionnés doivent être des **mots bien connus**. Dans le domaine orienté objet, ils doivent être largement acceptés par la communauté informatique.

- Les termes doivent avoir une signification **claire et précise**, facilement associable aux concepts qu'il couvre.

De plus, ils doivent être aussi distincts que possible, avec une signification très précise, afin de faciliter un lien direct entre le composant et le terme de classification correspondant.

- Les termes doivent être assez **généraux**, dans le sens qu'ils doivent recouvrir plus d'un terme spécialisé de la structure de classification. En d'autres mots, chaque terme doit être utilisé pour adresser plus d'un composant ou un ensemble de composants.

  Ainsi, une des tâches de base mais néanmoins très difficile du processus de classification est d'avoir un ensemble de termes généraux, suffisamment petit mais très expressif. Pourtant il serait bien plus simple d'avoir un espace de termes assez large mais cela impliquerait néanmoins des confusions et des performances de recherche assez mauvaises.

- Il faut éviter les redondances, c'est-à-dire que plusieurs termes différents d'une même hiérarchie ne doivent pas avoir une signification trop proche. Si tel était le cas, il serait préférable de les considérer comme synonymes.

3.3. Expérience de classification de logiciel.

Dans cette section, nous allons présenter un exemple de classification basé sur le cas suivant :

- classification de la classe " SET " de la librairie étendue C++ (APEX 1.0) [APE92].

Ce cas couvre des aspects intéressants de la classification. La librairie APEX a été écrite à FORTH (Crète) et contient la définition de composants C++. Durant ce processus, toutes les règles de base concernant le développement d'espaces de termes ont été observées. La sélection des termes appropriés, tâche qui s'avère bien souvent itérative, est l'un des aspects les plus importants de ce processus. De même, lors de l'insertion de nouveaux termes, il n'est pas rare que la re-classification de termes déjà ordonnés introduise une organisation sémantique plus correcte.

3.3.1. L'exemple traité.

Avant d'en arriver à la description détaillée de l'exemple de classification, nous allons d'abord introduire la classification proposée par une équipe d'informaticiens; classification contenant des mots clés. Elle nous servira de point de départ.

Cet exemple ainsi que les remarques qui s'y attachent tentent de clarifier les termes suggérés et définissent ensuite des principes généraux de classification. Il permet également de montrer comment on peut passer d'un système à mots clés vers le système de classification STA. Comme point de départ, on peut également se baser sur l'espace de termes développés lors de classifications précédentes.

Librairie : APEX 1.0 Extended C++ Standard Component

Class Set (ensemble)

Facettes et mots clés utilisés pour la Classification et la Recherche.

Application Domain	(Mots clés) UNIVERSAL_FOR_C++
Functionality	(Mots clés) COLLECTION HANDLING ITERATOR
Operates On	(Mots clés) GENERIC POINTER
Operations	(Mots clés) ALGEBRA CONSTRUCTOR_AND_DESTRUCTOR COPY_AND_ASSIGN ELEMENT_HANDLING ITERATOR LENGHT MEMBERSHIP RELATION PERFORMANCE_ANALYSIS STREAM_INSERTION

TYPE_MACRO

See also (Mots clés)

Bag

Bits

BitSet

Block

HashBag

HashMap

HashSet

HashSymbol

List

Map

Ordered_Collection

Pool

Sorted_Collection

Environment (Mots clés)

HP_720:HP-UX/c++2.1

HP_9300:HP-UX/C++2.1

IBM_RS6000:AIX/xLC++

SNI_MX300I:SINIX_5.41/C++2.1B

SNI_MX300N:SINIX_5.22/C++2.1

SNI_MX500I:SINIX_5.41/C++2.1B

SNI_PCD:SINIX_ODT_1.1/HCR_C++3.0

SNI_RW320:IRIX_4.01/C++2.1.1

SNI_WS30:DOMAIN_OS_10.3.5/C++2.1

SNI_WX200:SINIX_ODT_1.5/C++2.1B

SUN_SUN4:SUN_OS_4.1/C++2.1

3.3.2. Remarques générales sur la sélection des mots clés et leur correspondance dans STA.

Les catégories de classification décrites dans la liste ci-dessus, sont supportées par les facettes définies dans le modèle de classification STA de la manière suivante:

- Les informations concernant le domaine d'application **(application domain)** représentent des connaissances générales et communes pour tous les membres d'une librairie. De plus, ces informations doivent être indépendantes des spécificités telle que la langue des employés. Ainsi, le terme sélectionné " Universal for C++ " (universel pour C++) utilisé pour caractériser le domaine d'application de la librairie APEX ne semble pas adéquat. Un terme plus précis, tel que " Utility functions " (fonctions utilitaires) semble bien plus concret.

- Les informations concernant les fonctionnalités (**functionnality**) correspondent aux facettes STA **Operation**. Si un terme de cette catégorie exprime un comportement commun à un ensemble d'**Operations**, alors ce terme doit être localisé à un niveau supérieur dans le graphe de l'espace de termes des **Operations.**

- Les informations **Operates on** font directement références aux facettes STA **OperatesOn**. Les informations incluses dans cette catégorie mais qui ne sont pas considérées comme utiles pour la réutilisation ne font pas partie de l'espace de termes de cette facette.

- Les informations reprises dans la catégorie **See also** concernent principalement des classes similaires à celles que nous avons classées ou alors des concepts de l'espace de termes OperatesOn pertinents pour la classification des classes. Toutes ces informations

concernent un aspect général de la classification de la librairie C++ et ne sont pas nécessairement rattachées à chaque objet classé.

- Les informations concernant l'**environnement** sont directement associées aux facettes STA **Dependancies** . Elles décrivent les environnements sous lesquels les librairies sont exécutées et testées.

- Les **abstractions**, considérées comme des descriptions sémantiques de structures de données " actives ", sont rattachées aux classes C++ étendues.

- Les informations très détaillées n'apportant aucune contribution au mécanisme de réutilisation ne font pas partie des facettes.

- Les informations d'analyse statique ne sont pas incluses dans l'espace de termes des facettes.

- Les termes qui font trop référence au domaine étudié, et donc, pas très connu, ne sont pas inclus dans l'espace de termes des facettes.

- Les homonymes qui représentent des concepts différents ou peut-être inter-reliées, sont repris dans des hiérarchies différentes mais sémantiquement reliées.

- Dans l'espace de termes **Operation**, nous utilisons principalement des termes qui représentent le comportement (la fonctionnalité) et non l'objet qui exprime ce comportement.

- Ainsi, dans l'espace de termes **Operation**, nous sélectionnons principalement le verbe plutôt que le nom qui représente une activité spécifique. Si la forme du verbe est identique au nom correspondant et qu'il risque d'y avoir une confusion entre l'objet et l'action (p. ex. le verbe et le nom " Set "), alors le terme " operation " est ajouté.

3.3.3. Proposition de classification STA.

Ci dessous suivra la description détaillée du processus de classification de la classe "SET". La liste des termes de classification STA est comparée à celle suggérée précédemment et une explication du choix final est donné.

3.3.3.1. Classe : Set

Un SET (ensemble) est une collection homogène et non ordonnée d'éléments d'un type spécifique. Il n'y a pas de double dans un ensemble.

Abstraction

Set

Commentaire : le terme **SET** est générique et bien connu. En conséquent, c'est un terme qui convient bien.

OperatesOn

Les termes suggérés **GENERIC** (générique) et **POINTER** (pointeur) sont des informations concernant l'analyse statique. Ainsi, ils ne doivent pas faire partie de l'espace de termes **OperatesOn.**

Operation

- On utilisera le terme **SETOPERATION** (opération sur des ensemble) au lieu du terme suggéré **ALGEBRA** (algèbre).

Commentaire : le terme **SETOPERATION** a une signification plus claire et spécifique pour les opérations qu'il inclut. Dès lors, il a été choisi au lieu du terme **ALGEBRA**. Un autre terme **SETALGEBRA**, plus proche du terme **ALGEBRA** a été défini comme synonyme de **SETOPERATION**. Le terme **COLLECTION HANDLING** qui était inclus dans la catégorie "Functionnalities" (fonctionnalités) a été caractérisé comme étant plus générique que le terme **SETOPERATION** puisqu'il exprime un concept similaire (du moins pas trop différent) et qui, intuitivement, implique le rassemblement d'opérations (a set of operations).

- Le terme **CREATE-DESTROY** (créer - détruire) est utilisé au lieu du terme proposé **CONSTRUCTOR AND DESTRUCTOR** (constructeur et destructeur).

  Commentaire : Le terme **CONSTRUCTOR- DESTRUCTOR** est beaucoup trop lié au langage C++ et donc, pas très connu. Le terme **CREATE-DESTROY** a alors été sélectionné (notez qu'il est également sous forme verbale) et les termes **ALLOCATE, DELETE, INITIALIZE ET ASSIGN** sont directement associés à ce terme.

- Le terme **ELEMENT HANDLING** (manipulation d'éléments) n'a pas été inclus dans l'espace de termes **OPERATION**.

  Commentaire : les opérations incluses sous le concept ELEMENT HANDLING sont nombreuses : p. ex. GET, PUT, DELETE, etc. Il serait bien plus approprié d'organiser ces termes sous forme d'arbres sémantiques, puisqu'ils expriment la fonction de chaque opération et dès lors, seront plus utiles pour le processus de réutilisation.

- Le terme **ITERATE** (itérer) a été utilisé au lieu du terme **ITERATOR** (itérateur). La forme verbale a été sélectionnée puisqu'en général, on utilise le verbe qui exprime une "activité" (ici : une itération) et non le nom qui représente l'acteur (ex : l'itérateur).

Commentaire : dans l'espace de termes **OPERATION**, nous utilisons des termes qui expriment le comportement (fonctionnalités) et non l'objet qui exprime ce comportement. Ainsi, le terme **ITERATE** a été choisi au lieu du terme **ITERATOR**. C'est un terme précis et bien connu dans le domaine de l'ingénierie logicielle. C'est par conséquent un bon terme de classification.

- L'opération qui exprime l'appartenance à un ensemble est exprimée à travers le terme **BELONG** (appartenir), au lieu du terme proposé **MEMBERSHIP**.

  Commentaire : le terme **BELONG** semble plus approprié. Néanmoins, le terme **MEMBERSHIP** ainsi que le terme **CONTAIN** ont été utilisé comme synonyme.

- Le terme **COMPARE** (comparer) a été sélectionné pour regrouper les opérations utilisées pour évaluer les relations (égalités, inégalités et inclusions) entre objets. Le terme **RELATIONAL** (relationnel) a été utilisé au lieu du terme suggéré **RELATION** comme généralisation du terme **COMPARE**.

  Commentaire : le terme **RELATIONAL** a été choisi comme terme principal parce qu'il est plus général et il inclut en lui le concept de comparaison.

- Pour ce cas particulier, c'est-à-dire la classe **Set**, la fonctionnalité du processus **PERFORMANCE - ANALYSIS** (analyse de performances) est exprimée par le terme **MEASURE** (mesurer). Ce terme **MEASURE** a également été utilisé pour exprimer la fonctionnalité impliquée par le terme **LENGHT**.

  Commentaire : dans ce cas particulier, les actions effectuées lors du processus de **PERFORMANCE - ANALYSIS** sont principalement la prise de mesures.

- Le terme **INSERT** (insérer) est utilisé au lieu du terme **STREAM-INSERT** (insérer un champ).

Commentaire : l'action impliquée par le terme **INSERT** est indépendante de l'opérande. (p. ex. : stream).

- Le terme **TYPE-MACRO** couvre des informations d'analyse statique et donc, n'est pas inclus dans l'espace de termes **OPERATION**.

Dependancies

- Les **DEPENDANCIES** (dépendances) décrites dans la classe **SET** sont les mêmes pour toute la librairie. Les dépendances des objets C++ sont reprises dans la classification précédente sous le terme **ENVIRONMENT** (environnement). Elles font référence aux différentes plates-formes hardware, aux systèmes d'exploitations et aux langages de programmations.

Ces termes sont tous regroupés à un niveau supérieur sous les termes **HARDWARE, SYSTEME D'EXPLOITATION** (Operating System) et **LANGAGE DE PROGRAMMATION.**

3.3.3.2. Liste récapitulative de la classe SET en fonction du modèle de classification STA.

Abstraction SET

OperatesOn

Operation

- COLLECTION HANDLING
- SETOPERATION
- SETALGEBRA
- CREATE-DESTROY
  - ALLOCATE
  - DELETE
  - INITIALIZE
  - ASSIGN
- GET
- PUT
- DELETE
- ITERATE
- BELONG
- CONTAIN
- RELATIONNAL
- MEASURE
- INSERT

Dependancies

- HARDWARE
  - HP_720
  - HP_9300
  - IBM_RS6000
  - SNI_MX300I
  - SNI_MX300N
  - SNI_MX500I

SNI_PCD

SNI_RW320

SNI_WS30

SNI_WX200

SUN_SUN4

OS

HP-UX

AIX/xLC++

SINIX_5.41

SINIX_5.22

SINIX_ODT_1.1

IRIX_4.01

DOMAIN_OS_10.3.5

SINIX_ODT_1.5

SUN_OS_4.1

Programming language

C++2.1

C++2.1B

C++2.1.1

xLC++

HCR_C++3.0

3.4. Langage d'entrée de données dans SIB (Telos).

3.4.1. Introduction.

Le langage décrit ci-dessous est un langage qui nous permet d'entrer les données dans SIB. Il est dérivé du langage de représentation des connaissances TELOS. La structure des objets qu'il va traiter nous aidera à mieux comprendre le modèle de classification STA. Le but de cette partie est de donner une idée générale des caractéristiques du langage tel qu'il est implémenté dans SIB. Les détails concernant ce langage ainsi que la description de l'implémentation actuelle se trouvent respectivement dans [KMSB89] et [ITHACA92].

Le langage de représentation des connaissances TELOS est basé sur une structure orientée objet. Les objets de TELOS sont regroupés en "individuels" (entités, concepts, noeuds) et "attributs" (relations, liens d'attribut). Il fournit trois principes de structuration appelés la **classification** (instantiation inverse), la **spécialisation** (généralisation inverse) et l'**agrégation** (décomposition inverse) nous permettant d'obtenir une hiérarchie entre les différentes entités qu'on définit. L'utilisateur peut alors spécifier différents niveaux d'hiérarchies. Il peut ainsi définir son modèle à différents niveaux de telle manière à atteindre le niveau de précision requis.

La figure 3.4.1. nous présente un modèle du monde réel qui pourrait être décrit en Telos et où on retrouve trois niveaux d'hiérarchies de classification. Toutes les entités appartiennent à la classe **objet physique** qui se trouve au niveau de classification le plus haut. Les objets physiques peuvent être détaillés dans un niveau inférieur par les entités **livre**, **chapitres**, **introduction** et **conclusion**. Au niveau de classification le plus bas, on instancie les entités du niveau précédent. Sur ce schéma, nous pouvons également déterminer les relations de spécialisation qui existent entre l'objet **chapitre** et les sous-

objets **introduction** et **conclusion** ainsi que la relation d'aggrégation entre les objets **livre** et **chapitres** selon la relation **contient**.

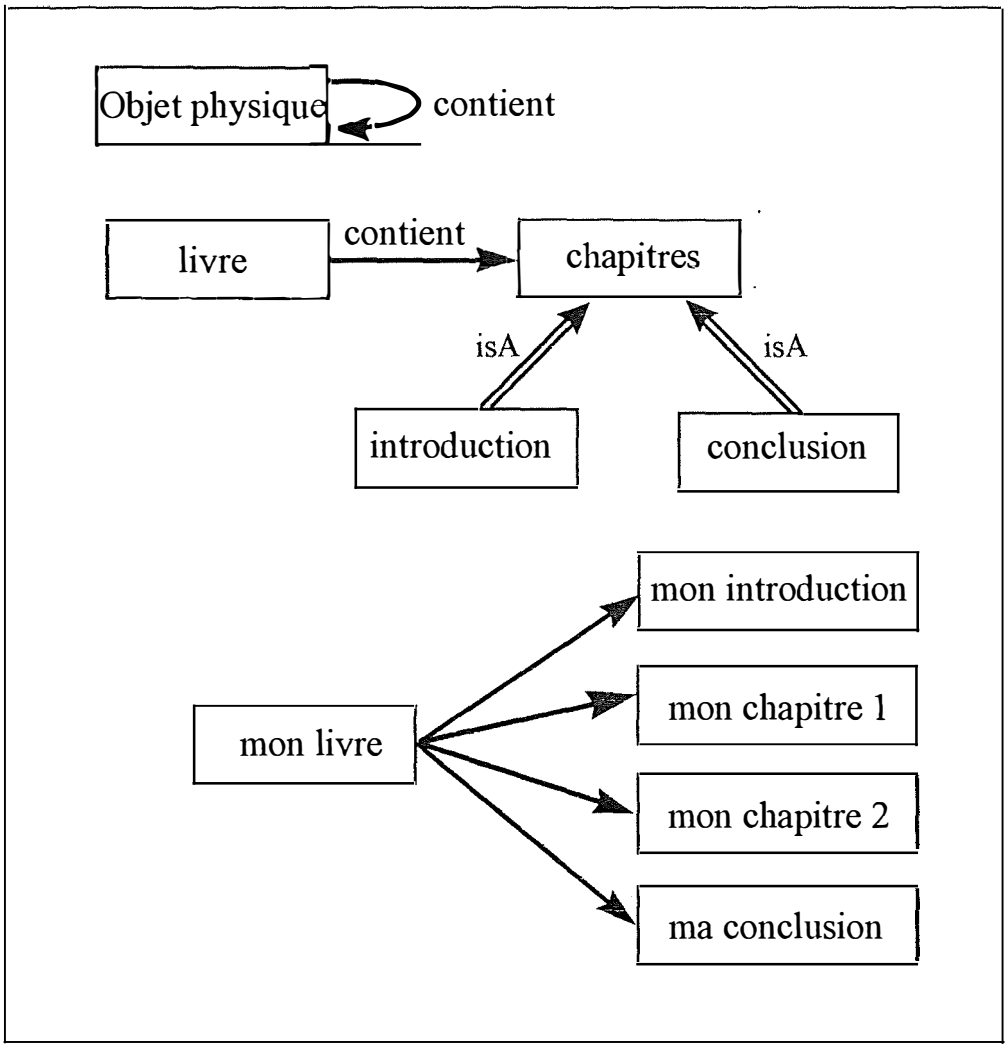

Figure 3.4.1 : Un modèle Telos.

A la suite de ce chapitre, nous allons d'une part détailler le langage de représentation de connaissances Telos et d'autre part, nous allons présenter les entités et relations relatives à la spécification du langage de programmation C++ (au niveau 2 de la classification) qui nous permettra d'instancier au niveau Token les objets (classes, méthodes, commentaires, etc) qui se trouvent dans nos programmes.

3.4.2. Concepts généraux sur le modèle de données.

Le SIB contient quelques classes prédéfinies décrites ci-dessous qui ne peuvent être modifiées par l'utilisateur. Elles constituent la population initiale et toute information entrée par l'utilisateur doit directement ou indirectement faire référence à celle-ci. Par la suite, nous noterons ces classes prédéfinies en *italique*.

Toutes les données qu'un utilisateur peut entrer sont regroupées en objets. Ce sont toutes des instances d'*objets*. Chaque objet a comme identifiant un nom logique. Les noms logiques sont des chaînes de caractères ASCII limitées actuellement à 95 caractères. Pour plus d'efficacité, ces noms sont représentés de manière interne par un identifiant (SYSID) qui est évidemment transparent pour l'utilisateur. Les objets de SIB sont des descriptions partielles des entités, relations, concepts correspondants ou des notions du monde réel. Les noms logiques doivent se rapprocher autant que possible des noms des objets du monde réel qu'ils représentent.

Parmi les objets, nous distinguons les **individuels** et les **attributs**. Les individuels correspondent à des choses réelles ou à des ensembles ou à des ensembles d'ensembles, etc. Les attributs correspondent aux relations entre les objets ou à des ensembles ou à des ensembles d'ensembles de relations. Les individuels sont des instances de *Individuel*. Ils n'ont pas de structure interne. Les attributs sont des instances de *Attribut*. Ce sont des objets à leur tour. Tout objet doit donc être un individuel ou un attribut. Ainsi par exemple, on peut écrire:

Individuel **isA** *Telos_Object*
Attribute **isA** *Telos_Object*

Un individuel avec un ensemble d'attributs ainsi que les objets avec lesquels ils sont en relation constituent un objet structuré (par exemple, comme une instance de classe C++ ou une classe C++). Actuellement, il n'y a pas de contrainte de cardinalité sur

l'ensemble d'attributs. De telles contraintes sont de moindre importance pour des propos descriptifs.

Les classes sont considérées comme un ensemble d'instances ayant un nom logique comme identifiant. Ces instances peuvent également être des classes. Les instances d'une classe doivent être déclarées explicitement. Il n'y a pas de classification automatique. Cependant, les classes peuvent n'avoir aucune instance. Les classes qu'un utilisateur peut définir sont des instances de *Classe*. *Classe* est partitionnée en *IndividualClass* et *AttributeClass* et :

Telos_Class **isA** *Telos_Object*
IndividualClass **isA** *Telos_Class*
IndividualClass **isA** *Individual*
AttributeClass **isA** *Telos_Class*
AttributeClass **isA** *Attribute*

Les individuels ou les attributs simples c'est-à-dire qui ne sont pas des classes sont des jetons (Tokens). Les "tokens" sont des instances de *Token*. On parlera de "token instantiation level". Les classes qui ont uniquement des jetons comme instances sont dites classes simples. Ce sont des instances de *S_Class* et on parlera de "simple class instantiation level". Les classes qui ont exclusivement comme instances des classes simples sont des metaclasses. Ce sont des instances de *M1_Class* et on parlera alors de "metaclass instantiation level". On peut continuer de la sorte pour enfin obtenir la hiérarchie d'instantiation de Telos (voir Figure 3.4.2). Actuellement, Telos peut supporter jusque *M4_Class* mais en pratique, il est difficile de trouver quelque chose au delà du niveau M3 qui aurait un sens. Ces niveaux de classe constituent une partition orthogonale des objets par rapport à la partition individuel-attribut. Un utilisateur peut instancier les intersections d'un niveau de classe avec soit un individuel soit un attribut.

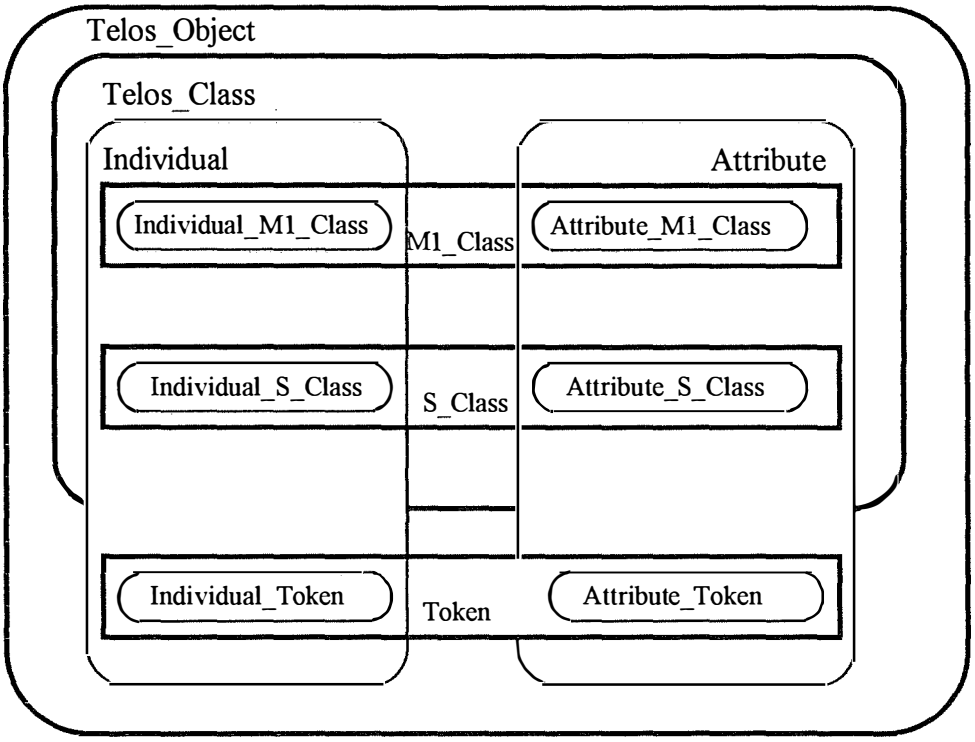

Figure 3.4.2 : Hiérarchie d'instantiation de Telos.

Exemple : `TELL Individual Personne in S-Class`
`end Personne`

Dans l'exemple précédent, on dit que " Personne " est une instance de la classe *individuel* (donc, fait partie de l'ensemble de gauche de la figure 3.4.2) et fait également partie de l'ensemble des *S_Class* c'est-à-dire que c'est une classe simple. Donc d'après le schéma, on fait référence à la classe prédéfinie : Individual_S_Class. On peut donc écrire que :

Personne est une instance de *Individual_S_Class* et
Individual_S_Class **isA** *Individual*
Individual_S_Class **isA** *S_Class*
Individual_S_Class **isA** *IndividualClass*
S_Class **isA** *Telos_Class.*

De même, trois classes simples *Individual* ont été définies pour les valeurs primitives : *Telos_Integer, Telos_Real et Telos_String*

3.4.2.1. Représentation interne des objets SIB.

Dans SIB, à l'exception des valeurs primitives, chaque individuel comporte les valeurs suivantes :

- SYSID - identifiant interne
- Sys_name - nom logique
- Sys_class - classe prédéfinie qui instancie l'objet.

ainsi que deux ensembles :

- IN_set - classes définies par l'utilisateur qui instancient l'objet.
- ISA_set - super-classe définie par l'utilisateur.

Dans SIB, chaque attribut comporte les valeurs suivantes :

- SYSID - identifiant interne
- Sys_name - nom logique
- Sys_class - classe prédéfinie qui instancie l'objet.
- Sys_from - l'objet en question
- Sys_to - l'objet en relation

ainsi que deux ensembles qui peuvent être vides :

- IN_set - classes définies par l'utilisateur qui instancient l'objet.
- ISA_set - super-classe définie par l'utilisateur.

Les ensembles IN_set et ISA_set implémentent des instantiations et héritages multiples. L'instantiation multiple est très utile pour la classification. Ainsi, par exemple, un composant logiciel " MyFifo " peut être une instance de " C++Class " et de

" FIFO_Implementation ". Les deux ensembles peuvent être vides. Les relations " IsA " ne sont pas supportées au niveau des " Token ".

3.4.2.2. Structuration avec les attributs.

Une classe attribut met en relation deux classes, Sys_from et Sys_to. De même, toutes les instances d'une classe attribut doit mettre en relation des objets qui sont eux-mêmes des instances des classes Sys_from et Sys_to (figure 3.4.3).

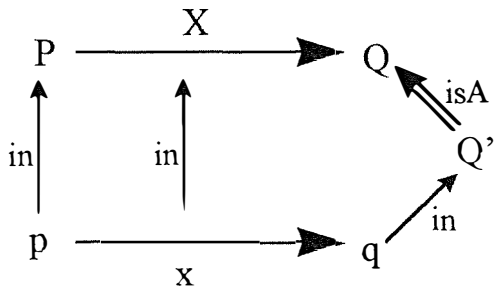

Figure 3.4.3.

L'objet p est une instance de l'objet P. L'attribut x de l'objet p est une instance de l'attribut X de l'objet P. L'objet q est une instance de l'objet Q. Les trois relations d'instantiation peuvent également être héritées une seule fois. Cette règle exprime le mécanisme de structuration des modèles de données orientés objets.

Un exemple équivalent écrit en C++ donnerait :

```
class P{
      class Q *X;
       } p;
 class Q q;
 p.X = &q;
```

Dans Telos, p.X peut être un ensemble. L'ensemble des éléments qui relie p à q a comme libellé[1] x. De même, p.X peut être vide. En outre, un attribut x peut être une instance de plus d'une catégorie[2]. Les objets qui sont relatés par des attributs peuvent être de différents niveaux. Le niveau de l'attribut lui-même doit être égal (par défaut) ou inférieur au niveau le plus bas des objets Sys_from et Sys_to.

3.4.2.3. Règles générales concernant la spécialisation.

Les relations IsA (spécialisation) doivent être déclarées explicitement par l'utilisateur. Les relations IsA ne peuvent pas être cycliques et sont transitives. Les classes qui ont entre elles une relation IsA doivent appartenir au même niveau d'instantiation. De même, si une classe P est une spécialisation d'une classe Q et Q étant une spécialisation d'une classe R, alors P est une spécialisation de R (figure 3.4.4).

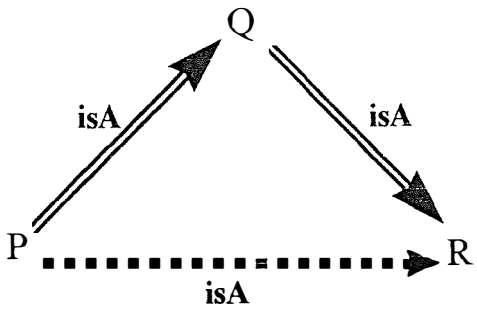

Figure 3.4.4

En outre, si une classe P est une spécialisation d'une classe Q et p est une instance de P, alors p est vu comme étant également une instance de Q (figure 3.4.5).

[1] Le nom logique d'un attribut peut également être appelé " libellé ".

[2] Les classes qui instancient un attribut peuvent également être appelées " catégories ".

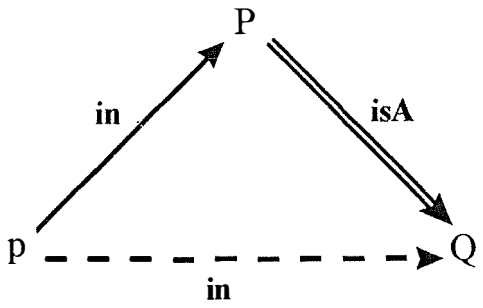

Figure 3.4.5

3.4.3. La commande TELL.

La commande TELL du langage Telos permet d'introduire des objets, des classes d'objets, des attributs et des classes d'attributs dans la base de données (SIB). Elle permet également de définir des relations hiérarchiques (ISA, INSTANCE) entre objets, classes d'objets, attributs et classe d'attributs. Telos fournit deux voies alternatives pour associer les attributs aux objets :

a. définition explicite utilisant la déclaration d'attributs.
b. définition implicite dans la déclaration des *Individuals*.

La définition explicite permet de définir tous les champs possibles d'un attribut (voir exemple à la section 3.4.3.2.). La définition implicite assigne " l'individuel " déclaré explicitement en tant que Sys_from à tous les attributs définis implicitement.

3.4.3.1. Nom des attributs.

Parmi les attributs ayant la même valeur Sys_from, les libellés des attributs doivent être uniques. Les libellés identiques ayant une relation IsA entre leurs valeurs Sys_from désignent une relation IsA entre ces attributs (relation IsA allant dans le même sens). De

même, les attributs définis au niveau des " Tokens " peuvent omettre le libellé (attributs sans nom). Ils sont identifiés en regardant la combinaison des noms de leurs valeurs Sys_from, Sys_to et IN_set.

Parfois, en raison d'une hiérarchie ou instantiation multiple, un simple nom n'est pas suffisant pour faire référence à une classe d'attribut.

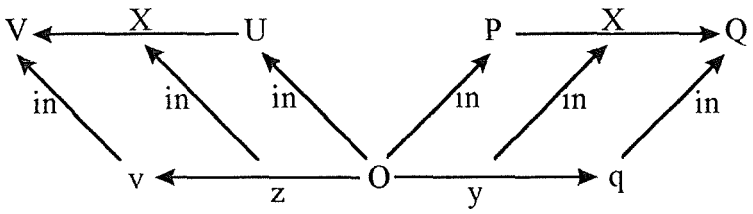

Figure 3.4.6

Dans l'exemple précédent, nous avons deux classes d'attribut ayant comme Sys_name X. L'attribut y est une instance de X avec X comme classe attribut de la classe P tandis que l'attribut z est une instance de X avec X comme classe attribut de la classe U. Pour résoudre cette dénomination conflictuelle, on complétera le Sys_name X avec le mot clé " from " suivi du Sys_name de l'objet qui instancie l'objet spécifié O (voir exemple ci-dessous).

TELL Individual O in S_Class, P, U **with**

X from P { ici, X est la classe d'attribut définie dans P }

y :q

X from U { ici, X est la classe d'attribut définie dans U }

z :v

end O

3.4.3.2. Exemples .

La liste des exemples ci-dessous démontre toutes les caractéristiques possibles du sous-système TELL de Telos. Chaque session TELL consiste en différentes transactions. Chaque transaction commence par BEGINTRANSACTION et se termine par ENDTRANSACTION et contient une opération TELL pour chaque objet Telos. Dans une transaction, la commande TELL est commutative. Les commentaires du code Telos se trouvent entre accolades {}. Les transactions suivantes décrivent comment on peut définir des objets Telos.

BEGINTRANSACTION
TELL Individual Persons **in** M1_Class **with**
attribute
familyRelation : Persons
end Persons

TELL Individual Person **in** S_Class, Persons **with**
familyRelation
fatherOf : Person;
motherOf : Person
end Person

TELL Individual LegalIdentity **in** S_Class
end LegalIdentity

TELL Individual ResIdentity **in** S_Class
end ResIdentity

TELL Individual Researcher **in** S_Class **isA** Person **with**

attribute

identity : ResIdentity

end Researcher

TELL Individual Citizen **in** S_Class **isA** Person **with**

attribute

identity : LegalIdentity

end Citizen

TELL Individual Status **in** S_Class

end Status

{ Ceci définit une classe d'attribut de la classe *Person* vers la classe *Status* ayant comme libellé *status* }

TELL Attribute status

components

from : Persons ;

to : Status ;

in S_Class

end status

TELL Individual studentStatus **in** Token, Status

end studentStatus

TELL Individual identity1 **in** Token, LegalIdentity

end identity1

{ *George* a comme catégorie d'attribut *status* qui est hérité de la classe *Person.*
L'attribut *identity* est issu de *Citizen* et est établi explicitement parce que *Researcher* a aussi le même attribut, c'est un cas d'héritage multiple.}

TELL Individual george **in** S_Token, Researcher, Citizen **with**
fatherOf
myFather : mike
status : studentStatus;
secStatus : employeeStatus
identity FROM Citizen
: identity1
end george

TELL Individual employeeStatus **in** Token, Status
end employeeStatus

{*Person* peut avoir ses propres instances en plus de celles de ses sous-classes }

TELL Individual mike **in** Token, Person
end mike

ENDTRANSACTION

Les exemples suivants illustrent l'utilisation des types TELOS (c'est-à-dire Real, Integer et String).

BEGINTRANSACTION

TELL Individual Researcher **in** S_Class **with**
attribute
name : String ;
salary : Integer ;
height : Real
end Researcher

TELL Individual Researcher1 **in** Token, Researcher **with**
salary
CSIsalary : 100000
height : 1.85
name : "george"
end Researcher1

ENDTRANSACTION

L'exemple suivant illustre l'utilisation de la clause FROM. Il cause une interruption de la transaction dû au fait que le token identity1 n'est pas une instance de la classe LegalIdentity bien qu'il devrait l'être car il existe une clause FROM dans la déclaration du token george spécifiant que l'entité associé à celui-ci via l'attribut identity doit être une instance de cette classe.

BEGINTRANSACTION

TELL Individual Researcher **in** S_Class **with**
attribute
identity :ResIdentity
end Researcher

TELL Individual Citizen **in** S_Class **with**
attribute
identity :LegalIdentity
end Citizen

TELL Individual LegalIdentity **in** S_Class
end LegalIdentity

TELL Individual ResIdentity **in** S_Class
end ResIdentity

TELL Individual george **in** Token, Researcher, Citizen **with**
identity from Citizen
: identity1
end george

TELL Individual identity1 **in** Token, ResIdentity
end identity1
ENDTRANSACTION

Les exemples suivants illustrent la création de relations IsA entre des classes d'attributs déclarées explicitement.

BEGINTRANSACTION

TELL Individual ResIdentity **in** S_Class **isA** PersonIdentity
end ResIdentity

TELL Individual Person **in** S_Class
end Person

TELL Individual Authority **in** S_Class
end Authority

TELL Attribute identity
components
from : Person
to : PersonIdentity
in S_Class **with**
attribute
certifiedBy : Authority
end identity

TELL Individual Researcher **in** S_Class **isA** Person
end Researcher

TELL Individual AcadAuthority **in** S_Class **isA** Authority
end AcadAuthority

TELL Attribute resIdentity
components

from : Researcher

to : ResIdentity

in S_Class **isA** identity from Person **with**

attribute

certifiedBy : AcadAuthority

end resIdentity

TELL Individual PersonIdentity **in** S_Class

end PersonIdentity

ENDTRANSACTION

L'exemple suivant illustre la déclaration explicite des attributs TOKEN.

BEGINTRANSACTION

TELL Individual Person **in** S_Class

end Person

TELL Individual Authority **in** S_Class

end Authority

TELL Attribute identity

components

from : Person

to : PersonIdentity

in S_Class **with**

attribute

certifiedBy : Authority

end identity

TELL Individual PersonIdentity **in** S_Class

end PersonIdentity

TELL Individual george **in** Token, Person
end george

TELL Individual identity1 **in** Token, PersonIdentity
end identity1

TELL Individual authority1 **in** Token, Authority
end authority1

TELL Attribute myIdentity
from : george
to : identity1
in Token, identity **with**
certifiedBy : authority1
end myIdentity

ENDTRANSACTION

3.4.4. Modèle Telos pour l'analyse statique de programmes C++.

A la section précédente, nous avons présenté les caractéristiques du langage Telos. Nous allons donner ici une description générale du modèle de représentation de connaissances pour l'analyse statique de données se trouvant dans les programmes C++. Afin d'éviter une étude trop technique, le modèle détaillé se trouve en annexe I.

Ce modèle doit contenir des informations utiles pour l'analyse statique. Dans un programme C++, de telles informations pourraient être les classes, fonctions, fonctions membres, des variables statiques et externes, les fichiers, les types et leurs relations, les commentaires. Il n'y aucun intérêt à représenter des variables locales ou tout autre information locale car d'une part le modèle serait trop complexe et d'autre part, si quelqu'un recherche des informations spécifiques à un programme, il serait préférable de consulter le code correspondant. De même, les informations sont extraites des programmes C++ en utilisant un « parser ».

Le langage Telos supporte les hiérarchies d'instantiation et de classification (IsA). Ces hiérarchies sont représentées par des liens entre les différents objets. Avec les liens de type **Instance**, les objets sont regroupés en classes et avec les liens de type **IsA**, les classes sont caractérisées comme sous-classes ou super-classes d'une classe particulière. Afin de rester consistant avec la terminologie C++, à la suite de cette section, les termes classes, super-classes et sous-classes seront remplacés par Type, Super-type et Sous-type.

Pour le modèle C++, nous utilisons trois niveaux d'instantiation (voir figure 3.4.7). Au niveau des S-Class, nous allons modéliser la construction du langage C++. C'est équivalent à la génération de schéma pour une base de données. Les données qui sont extraites de nos programmes sont enregistrées comme objet au niveau Token (jeton). Chaque objet du niveau Token est relié avec son type correspondant au niveau S_Class

via des liens d'instantiation. Chaque objet peut avoir plus d'un type, ce qui signifie qu'on peut utiliser plus d'un lien pour connecter cet objet particulier aux différents types correspondants (instantiation multiple).

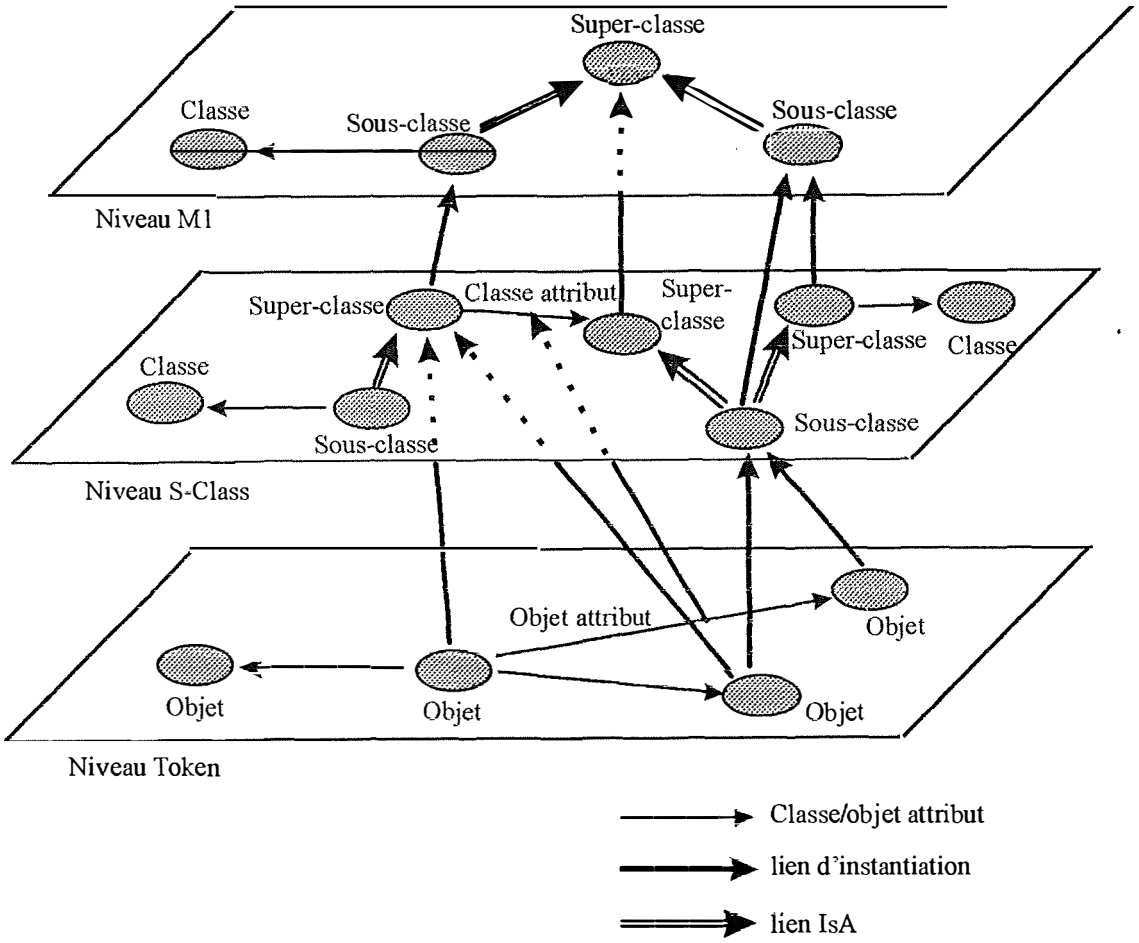

Figure 3.4.7 : Niveaux d'instantiation du modèle de données.

De la même manière que nous modélisons les constructions du langage C++, au niveau S_Class, le niveau M_Class contiendra des constructions générales sur les langages de programmations. Ce sont des constructions abstraites tels que des objets, méthodes, fonctions, commentaires qui existent dans une variété de langages de programmations orientés objets. Les constructions du langage C++ ne sont qu'une instantiation des constructions du niveau supérieur. Ainsi, le Pascal Orienté Objet, le langage Cool ou n'importe quel autre langage orienté objet pourrait être une instance des constructions générales des langages de programmations du niveau M_Class.

3.4.4.1. La hiérarchie IsA.

La hiérarchie IsA représente la relation de spécialisation entre objets de même niveaux. Un type spécifique peut avoir soit un super-type, soit un sous-type. Une instance d'un type est également une instance de son super-type. Ce principe est très utile pour l'entrée des données car il nous aide à identifier le type d'un objet même si nous n'avons pas encore suffisamment d'informations pour l'identifier. La seule information que nous connaissons étant le super-type de son type. Nous pouvons ainsi utiliser ce super-type jusqu'au moment où nous aurons suffisamment d'information pour identifier le type correct de cet objet.

Une autre caractéristique de la hiérarchie IsA est l'héritage. En d'autres mots, un type hérite des types d'attributs de ses super-types. Voyons la figure suivante (3.4.8) :

Le type **RoutineSignature** a les types d'attributs suivants : **arguments** et **return_type.**

Le type **Routine** est un sous-type du type **RoutineSignature**. Dans ce cas, le type **Routine** hérite des types d'attributs de son super-type **RoutineSignature** mais possède également ses types d'attributs propres. Ainsi, le type **Routine** possède les types d'attribut suivants : **arguments, return_type, signature, calls, defined_in, line_no**. Le type **MemberFunction** n'a pas de type d'attribut particulier mais hérite des type d'attribut du type **Routine**.

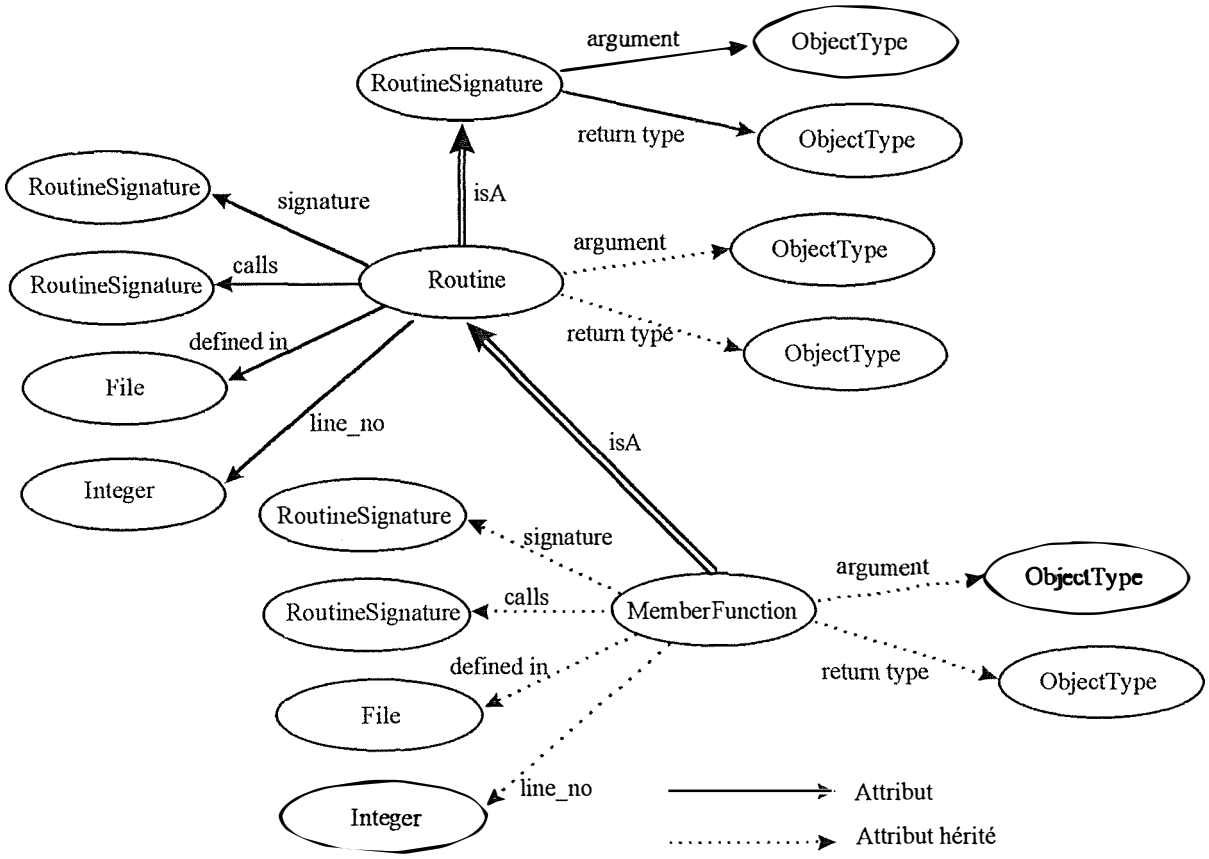

Figure 3.4.8. : Héritage dans une hiérarchie IsA.

Ce style de modélisation permet à SIB d'avoir une structure de données très proche des règles syntaxiques du langage C++. La **RoutineSignature** est le lien interne vers une **Routine** particulière. Comme instance de la **RoutineSignature**, nous allons considérer le nom de cette routine avec ses types d'attributs spécifiques et sa valeur de retour. Une **Routine** est donc une spécialisation de **RoutineSignature**.

Exemple : Fichier test.c

```
int multiplication(int a,float b) ;          //Routine Signature
void processus()                             // Routine
{
:
multiplication(...) ;                        // Routine call
```

```
:
}
int multiplication(int a, float b)                    // Routine
{
:
}
```

Dans une hiérarchie IsA, la spécialisation nous aide à distinguer une Routine d'une RoutineSignature (sa déclaration). Ceci est très utile car une déclaration peut avoir plus d'une implémentation et les appels de routines se réfèrent uniquement aux déclarations et non aux implémentations.

3.4.4.2. Exemple simple.

Cet exemple illustre un modèle simple ainsi que la manière utilisée pour extraire l'information utile pour l'analyse statique. Imaginons donc que nous avons un programme C++ contenant la déclaration suivante : *int pop(int *, int b) ;*

Avant même que le parser lise le programme source C++ , SIB contient le schéma au niveau S_Class ainsi que quelques objets triviaux au niveau Token (sur la figure suivante, la ligne pointillée sépare le niveau S_Class du niveau Token). Au niveau S_Class, on trouve les types **Declarator** et ses sous-types **RoutineSignature** et **ObjectType**. Le type **RoutineSignature** a un sous-type **FunctionSignature** et le type **ObjectType** a deux sous-types **FondamentalType** et **C++Pointer.** Le type **FunctionSignature** a deux types d'attributs, **arguments** et **return_type** qui pointent vers le type **ObjectType** et qui expriment le fait qu'une routine a des arguments et une valeur de retour. Le type **C++Pointer** a un type d'attribut, le pointeur **(point)** qui pointe vers le type **ObjectType** et qui exprime le fait qu'un pointeur pointe vers un type particulier. L'objet **int** est une instance du type **FundamentalType** et l'objet **int*** est une instance du type **C++Pointer.** L'objet **int*** a un objet attribut qui pointe vers **int** et qui est une instance du type d'attribut **points**.

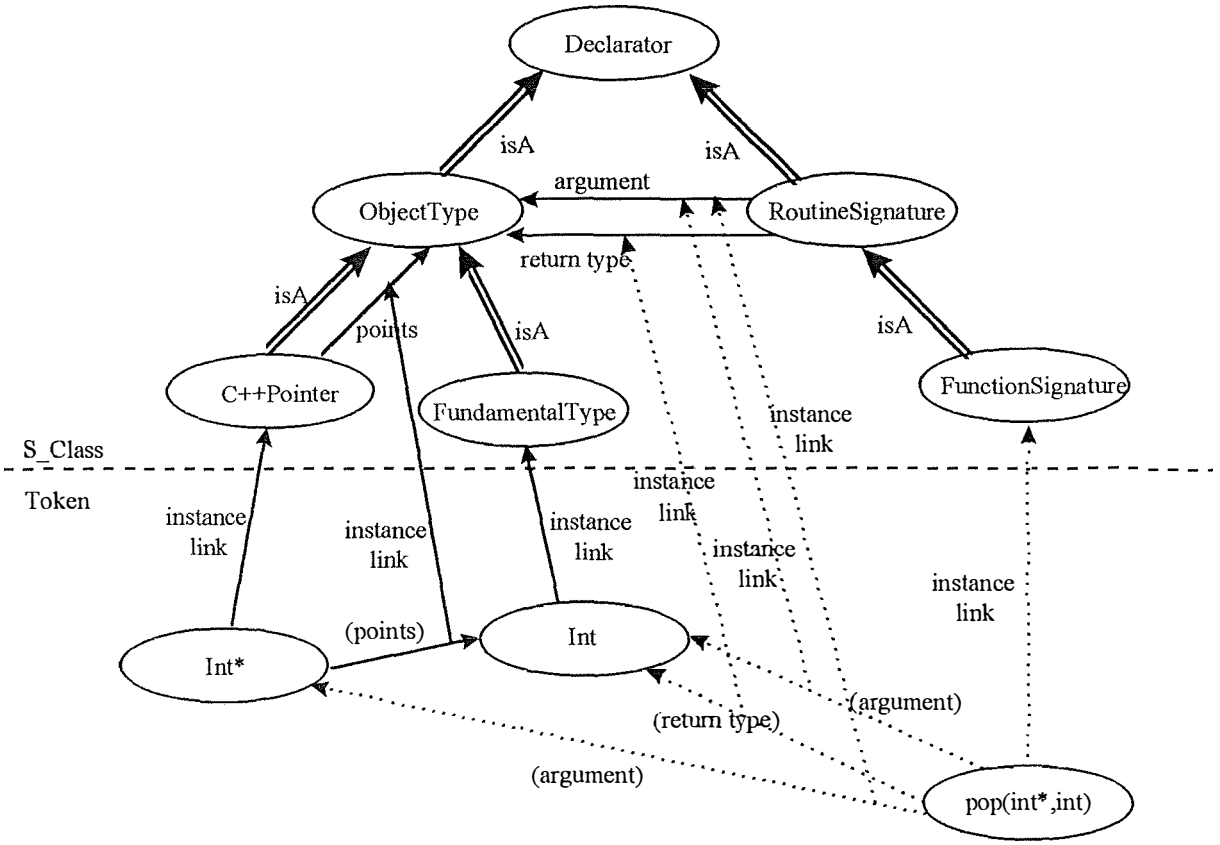

Figure 3.4.9. : Un exemple simple de modélisation C++.

Le parser lit le programme C++ et produit quatres objets. (Ces objets ainsi que les liens d'instanciation figurent sur le schéma en pointillés). L'objet **pop(int *,int)** est une instance du type **FunctionSignature**. Les trois autres sont des objets attributs. Parmis ceux-ci, deux sont des instances du type d'attribut **argument** et le dernier est une instance du type d'attribut **return-type**.

Nous avons ainsi présenté le principe général de spécification des objets C++ en Telos. La section suivante va présenter la classification des objects C++. Nous n'allons cependant pas entrer dans les détails de cette classification, l'objectif de ce chapitre étant de présenter le principe de spécifications en Telos et non pas la présentation détaillée des objets du langage C++.

3.4.4.3. Détail de la classification des object C++.

Le schéma développé pour l'analyse statique comporte trois parties. La première concerne la description de toute déclaration qu'on peut trouver dans un programme C++. La seconde partie décrit les objets (variables) qui sont créés lorqu'un programme tourne. La troisième partie décrit le système de fichier, la librairie et le système de version.

a) Déclaration.

Un déclarateur déclare un objet simple, une fonction ou un type. Dans la figure suivante, nous pouvons voir le type **déclarator** et ses spécialisations.

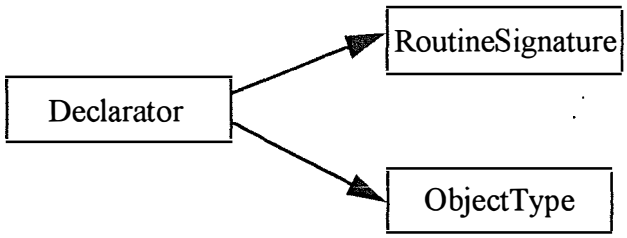

Figure 3.4.10.: Arbre de classification du type Declarator (déclarateur).

Le type declarator est le type général de tous les déclarateurs et comme vous pouvez le voir, il a deux spécialisations, l'**ObjectType** et le **RoutineSignature**. L'**ObjectType** est le type général de tous les déclarateurs qui déclarent des objets et le **RoutineSignature** est le type général de toutes les déclarations qui déclarent des routines.

Ci-dessous, nous pouvons voir les arbres de classification des types **ObjectType** et **RoutineSignature**.

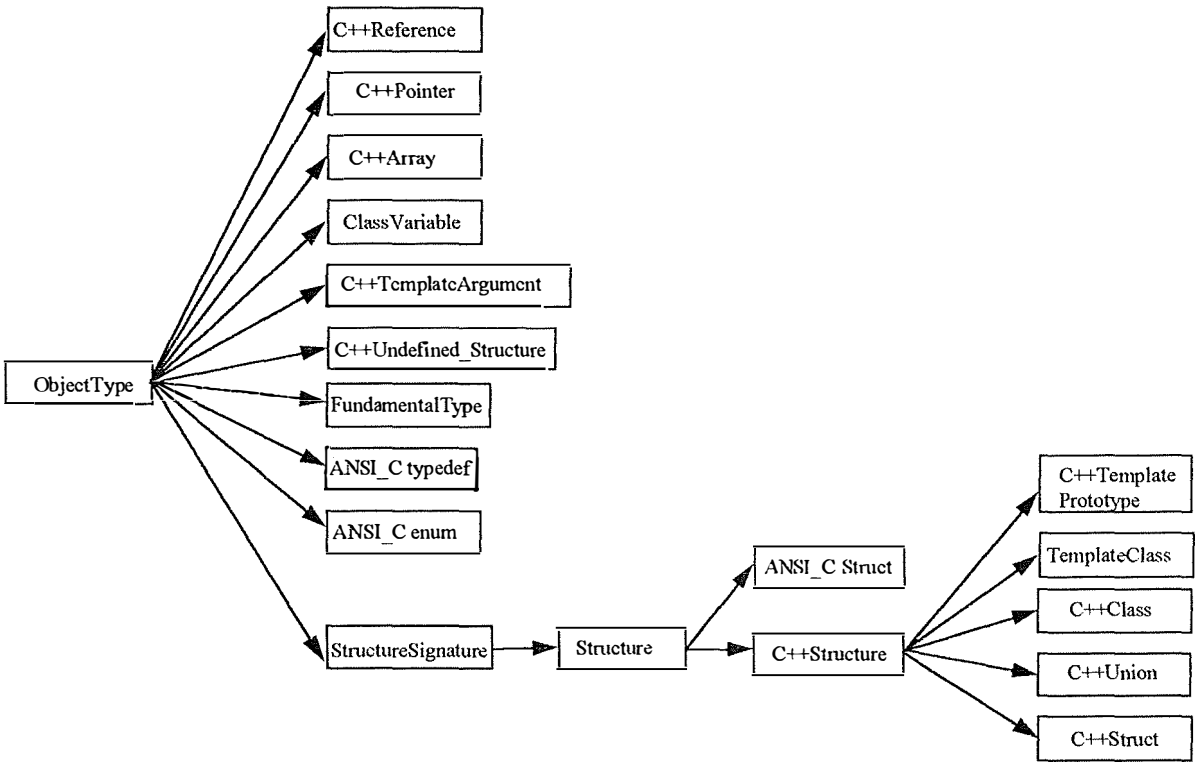

Figure 3.4.11.: Arbre de classification du type ObjectType.

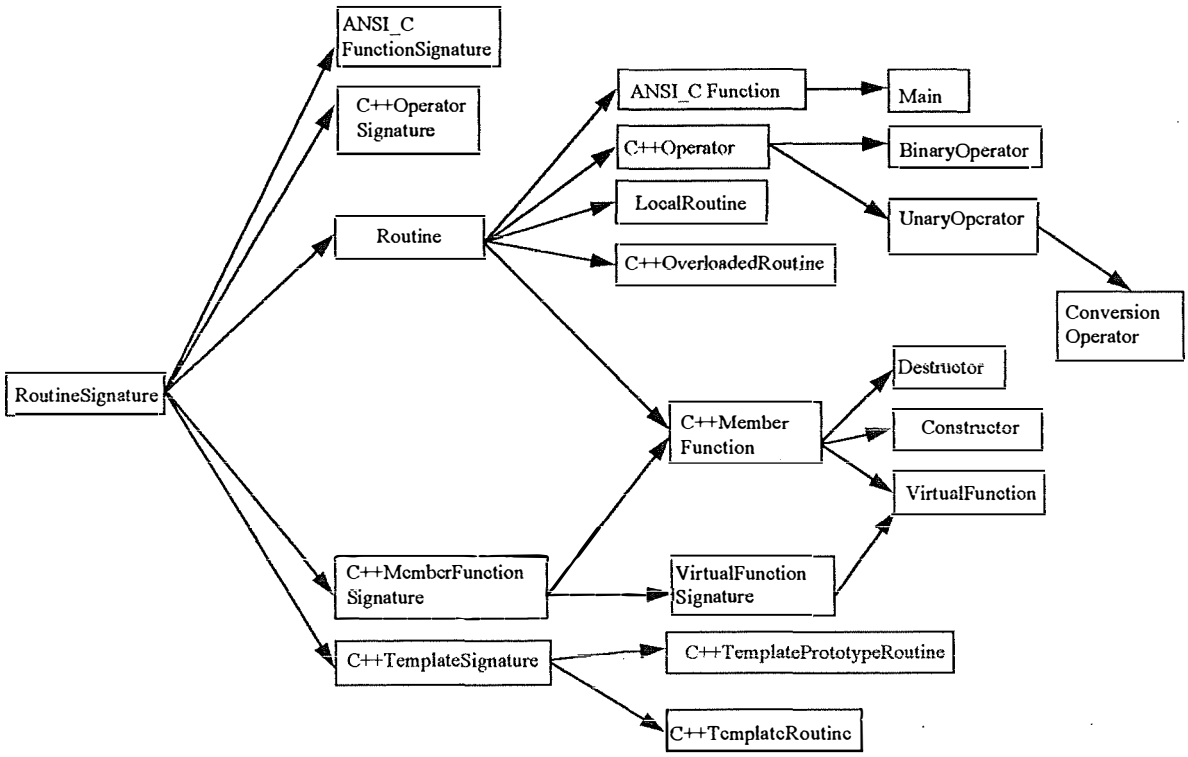

Figure 3.4.12.: Arbre de classification du type RoutineSignature.

b) Variables.

Dans les programmes C++, nous sommes intéressés par trois types de variables: le type **ConstVariable** (variables constantes), le type **LocalVariable** (variables statiques) et le type **GlobalVariable** (variables globales, accessibles par tous les modules). Nous n'allons pas produire d'analyse statique des données qui se trouvent dans les classes, structures ou fonctions car ces variables ne sont pas visibles à l'extérieur de l'objet.

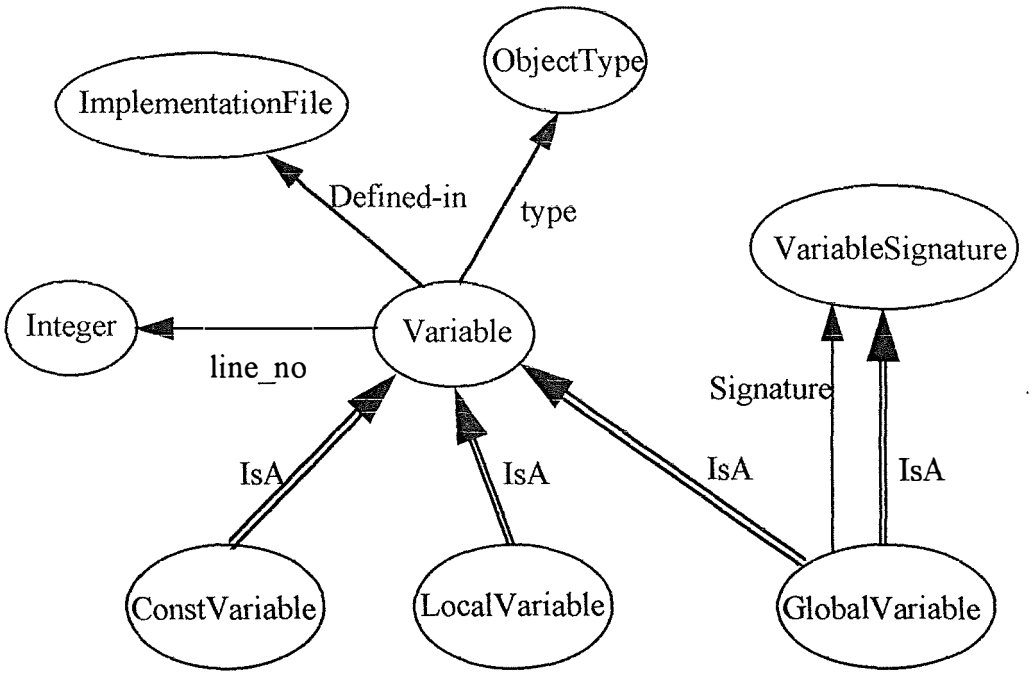

Figure 3.4.13. . Sous modèle des variables.

c) Gestion du code source.

La troisième partie du modèle concerne la gestion des fichiers des programmes sources. Les deux concepts principaux sont les fichiers et la collection de fichiers. Par fichier, nous entendons les fichiers exécutables, les fichiers d'archivage et les fichiers sources. Nous illustrons ci-dessous le modèle complet de la gestion du code source.

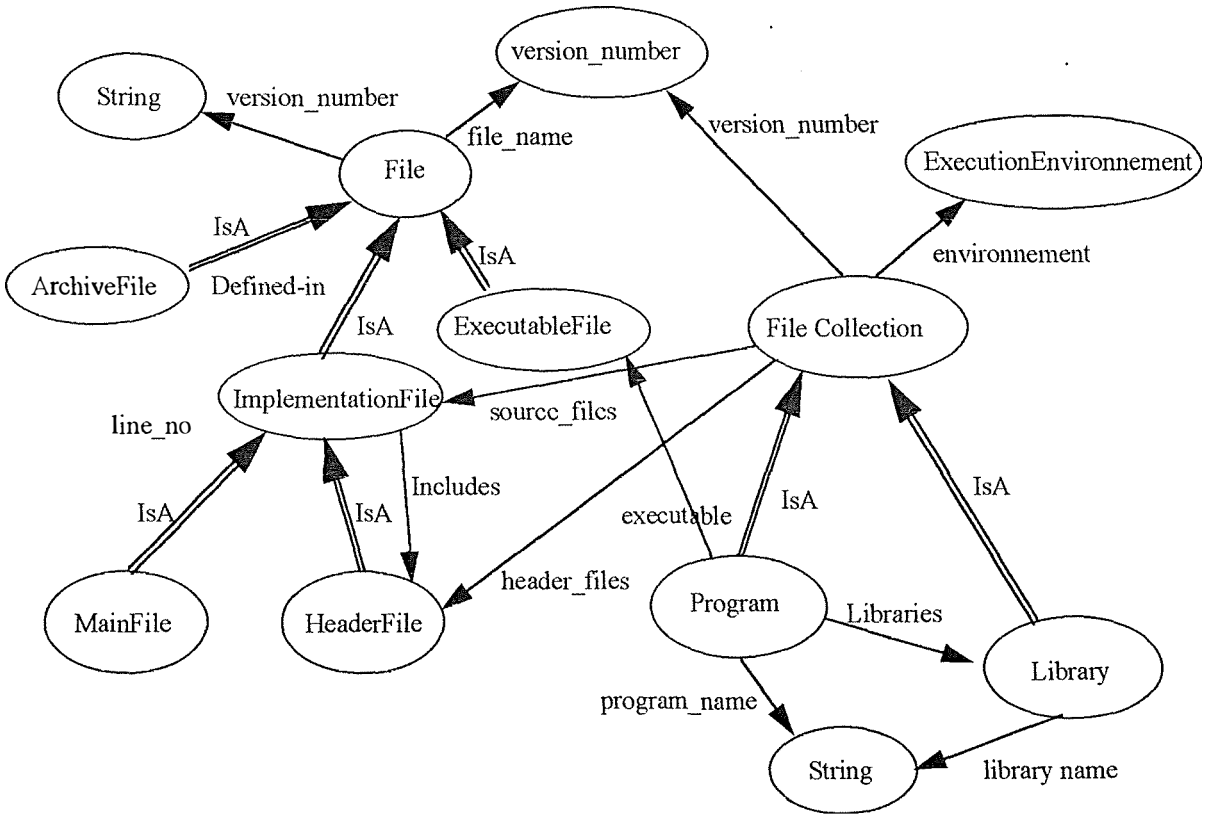

Figure 3.4.14. Sous-modèle pour la gestion des fichiers sources.

4. COMMENTAIRES EN LIGNE.

4.1. Introduction.

Les chapitres précédents ont été consacrés à l'analyse d'un modèle de classification de composants C++, à la présentation d'un langage de représentation de connaissances (TELOS) utilisé pour décrire ce modèle et à l'interface graphique de SIS pour afficher le résultat de la classification des composants C++. Ce chapitre va présenter un automate qui sera capable de lire des programmes sources et en dégagera les commentaires à associer aux composants réutilisables. L'apport des outils présentés dans les chapitres précédents nous permettra d'afficher ces commentaires graphiquement. Nous parlerons alors de **commentaires en ligne**.

Ainsi, avant de détailler le processus de classification des commentaires, il s'impose d'introduire tout d'abord cette notion qui a été empruntée d'une expression plus générale communément utilisée lors des problèmes de documentation : la " documentation en ligne ".

4.2. Documentation en ligne.

4.2.1. Introduction.

Il est à noter que depuis des années, l'information qu'on perçoit n'arrête pas de croître. L'information scientifique double tous les cinq ans. Seulement, pour toute l'information écrite, seule une minorité est lue. Seuls dix pour cent de tous les écrits sont publiés et parmi ceux-ci, moins de dix pour cent sont lus. La solution pour gérer ce déluge d'information serait d'utiliser les systèmes informatiques. Il faut donc concevoir un système qui fournit l'information désirée sous une forme aussi claire que possible.

4.2.2. Qu'est-ce que la documentation en ligne?

La documentation en ligne ne connaît pas une seule définition bien précise. La plupart des autorités acceptent le fait que la documentation en ligne inclut l'information affichée sur un écran d'ordinateur mais leurs avis divergent sur d'autres points. Certains, comme Henrietta Shirk limitent l'utilisation du terme documentation en ligne à l'information présentée par l'ordinateur telle que les bases de données bibliographiques ou les journaux électroniques [Shirk88]. Seulement, il reste encore à déterminer quelle partie de l'interface est considérée comme documentation en ligne. Henrietta Shirk estime qu'il faut inclure :

"*les messages d'erreur, les messages d'aide, les guides de références, et les* ***tutorials*** *qui apparaissent sur l'écran plutôt que sur papier* ".

Dans leur définition, Diana Patterson et Paul Evitts font la relation entre les documents en ligne et les documents sur papier :

"la documentation en ligne est une information qui est introduite dans un ordinateur et qui est consultée à l'aide de cet ordinateur plutôt que de l'imprimer ou de la lire dans un livre".

La documentation en ligne englobe une riche diversité, du simple message jusqu'au programme qui regrouperait différents documents déjà écrits. Elle inclut l'utilisation de l'ordinateur pour communiquer de l'information qui pourrait se trouver sur papier mais inclut également différents types et organisations de documents qui ne sont pas possible sur papier. De même, elle introduit des médias supplémentaires tels que l'animation, la musique, la voix et la vidéo. La documentation en ligne utilise donc l'ordinateur pour communiquer l'information sans regarder au format et ne se soucie pas de voir si l'information existe également sous d'autres formes. Peut-être aurait on pu choisir le terme " communication électronique " mais le terme " documentation en ligne " semble définitivement adopté.

4.2.3. Il y a l'information et l'accès à celle-ci.

Les systèmes de documentation en ligne ont deux composants essentiels. Le premier est **l'information** enregistrée de manière électronique. La plupart du temps, c'est du texte mais comme on l'a dit, on y trouve également d'autres médias. Le second composant essentiel est le **moyen pour accéder** à cette information. Toute l'information enregistrée de manière électronique n'est pas de la documentation en ligne. La documentation en ligne nécessite un moyen pour les utilisateurs de trouver et d'afficher rapidement les informations qu'ils recherchent.

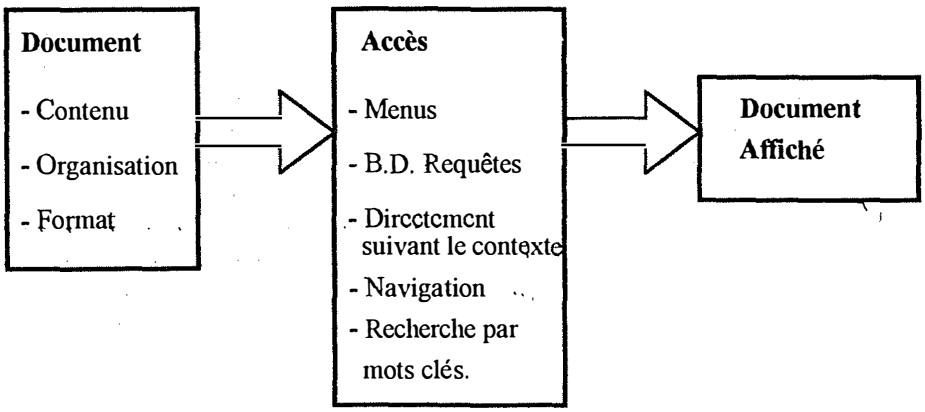

Figure 4.1. : Système de documentation en ligne. [Horton90]

Ces deux composants sont intimement liés (figure 4.1.). Les moyens d'accès à l'information doivent utiliser cette structure pour permettre à l'utilisateur d'accéder le plus rapidement possible à une unité d'information.

De même, les documents en ligne et sur papier contiennent trois types d'information : le **contenu**, le **format** et **l'organisation**. Le **contenu** se réfère aux mots, aux images et aux sons qui forment le sujet du document. Le **format** contrôle comment le contenu est affiché. Il inclut la typographie, la mise en forme et les couleurs. L'**organisation** spécifie l'ordre dans lequel les éléments sont présentés et les relations qui existent entre eux.

Ce qui différencie les documents en ligne des documents sur papier, c'est que sur une page imprimée, le contenu, le format et l'organisation sont liés. On ne peut en changer l'un sans affecter l'autre ou détruire l'oeuvre. Par contre, les documents en ligne séparent totalement le contenu, le format et l'organisation et le concepteur doit gérer chaque élément séparément.

4.2.4. Que peut offrir la documentation en ligne?

En utilisant une information stockée de manière électronique, la documentation en ligne permet:

- **une distribution rapide de l'information :** La documentation en ligne étant distribuée de manière électronique, les mises à jour sont distribuées pratiquement simultanément (via des réseaux) et les utilisateurs peuvent rapidement disposer d'une information récente.

- **l'accès immédiat aux informations désirées :** Avant de lire, les utilisateurs doivent chercher ce qu'ils veulent lire. De manière traditionnelle, les novices et les expérimentés savent lire aussi bien l'un que l'autre mais les experts ont l'avantage de trouver plus rapidement l'information désirée. Avec le système de documentation en ligne les inexpérimentés peuvent trouver l'information recherchée aussi vite que les utilisateurs expérimentés le font avec les documents sur papier. Ce système évite aux utilisateurs les longues recherches qu'ils connaîtraient s'ils devaient se rendre dans une librairie. Ainsi, un bon système de documentation en ligne nous permet d'éviter l'inconvénient le plus significatif que nous connaissons avec les livres, à savoir, la longue recherche nécessaire avant de trouver une information.

- **d'intégrer l'information avec le produit:** le problème avec les manuels de références, c'est qu'ils ne sont pas conçus et développés comme faisant partie intégrante du produit mais sont souvent développés ultérieurement. Le résultat est évident: aucun utilisateur ne semble satisfait de la documentation même s'il a fallu un temps considérable pour la réalisation. De plus, combien de fois ne va-t-on pas consulter un manuel sans jamais le trouver parce qu'il a été emprunté. La documentation en ligne évite donc ce problème en intégrant le manuel au produit. Les utilisateurs ne considèrent plus cette documentation comme quelque chose qui a été ajoutée au produit mais plutôt comme une partie du même produit.

De plus, puisque le manuel de référence est considéré comme faisant partie intégrante du produit, un effort particulier est consacré pour son développement. Les programmeurs ou ingénieurs donnent beaucoup plus d'importance à sa réalisation qu'ils n'en donne pour la réalisation de la documentation sur papier. Bien qu'ils la considèrent utile, elle ne fait tout de même pas partie des priorités.

- **un meilleur support du produit:** Janet Walker cite trois attitudes courantes à propos des documentations [Walker88].

1. Les projets de documentation ne sont pas prévus dans le planning et dans le budget et dès lors, la qualité du résultat final est décevant.

2. Tout le monde est conscient des problèmes de la documentation et tous essayent de l'éviter

3. Personne ne veut vraiment écrire de documentation technique; c'est la tâche ennuyeuse que l'on effectue après que le plus dur travail soit terminé.

Intégrée dans le développement du produit, la documentation en ligne peut être un remède à ces problèmes.

Bien qu'il soit difficile de quantifier le bénéfice que nous apporte la documentation en ligne, on remarque néanmoins des économies favorables dans les coûts de production. On épargne du temps lors de l'apprentissage, lors de la recherche d'information et lors de la correction d'erreurs. La documentation en ligne ne vient pas remplacer les documents sur papier mais limite néanmoins notre dépendance à ceux-ci.

4.3. Outil de génération de commentaires en ligne.

4.3.1. Objectif.

Cette section va présenter une méthode qui permettra de générer des commentaires en ligne à partir des commentaires des programmes sources écrits en C++. L'idée est de pouvoir dégager des commentaires les propriétés des composants classifiables. De même, comme on l'a vu à la section précédente, ces commentaires en ligne doivent contenir trois types d'information : le **contenu**, **l'organisation** et le **format**. Retrouvons-nous ces trois éléments dans notre solution?

La figure 4.2. représente le schéma logique de la solution adoptée pour résoudre ce problème. Tout d'abord, il s'agit de construire un automate, le *Parser*, qui lirait les programmes sources et en dégagerait des commentaires. Or, ce qui nous intéresse, ce n'est pas d'extraire tous les commentaires des programmes mais uniquement l'information utile pour la classification. Il faut donc pouvoir identifier les commentaires à extraire relatifs aux composants classifiés. De même, pour chaque composant réutilisable, notre solution propose différents types de commentaires qui seront identifiés dans les programmes par des mots clés. En lisant les programmes sources, le *Parser* n'aura qu'à identifier ces mots clés et en dégager l'information nécessaire à associer aux composants réutilisables (voir section 4.3.2.).

Cependant, pour que le *Parser* puisse identifier les différents commentaires et les associer correctement aux composants classifiables, l'écriture de ces commentaires doit respecter certaines règles syntaxiques. En effet, puisqu'il existe des relations entre les commentaires, le noeud du problème consiste à définir une grammaire appropriée (voir détails à la section 4.5.2). Par conséquent, lors de l'écriture des commentaires, le programmeur devra respecter cette syntaxe.

Une fois qu'on a dégagé les commentaires des composants classifiés, nous avons obtenu le premier élément de notre documentation en ligne, à savoir le **contenu**.

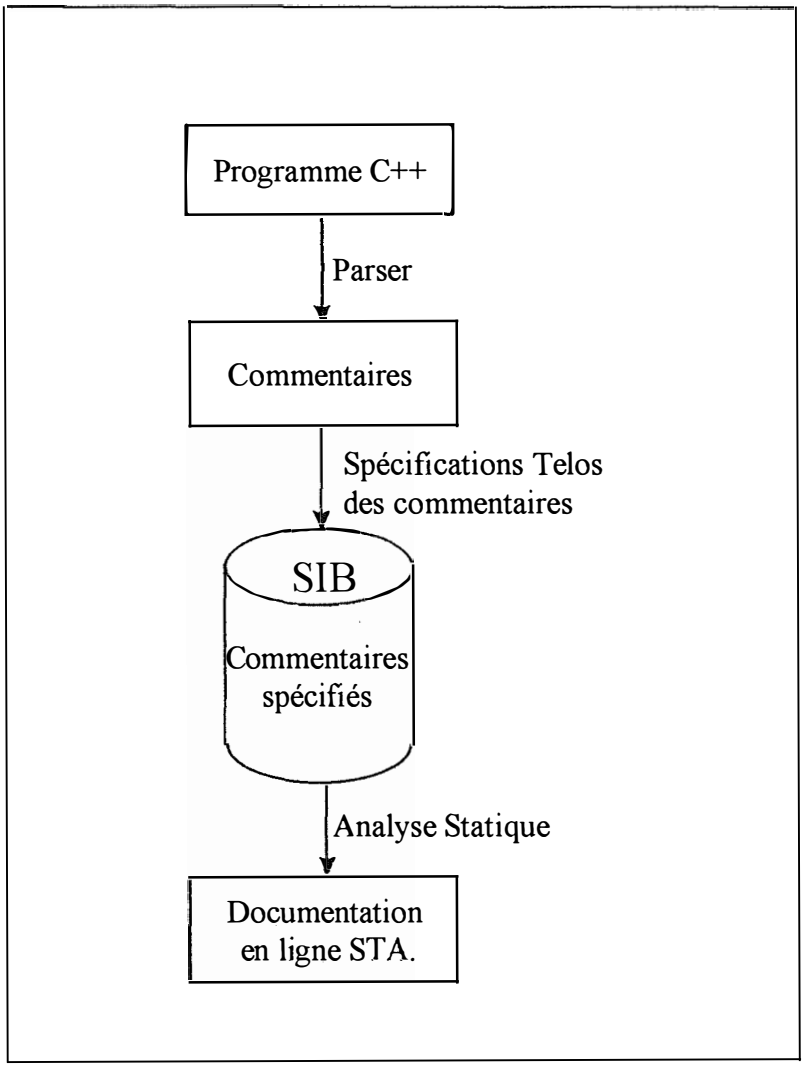

Figure 4.2.: Schéma logique de la solution.

L'étape suivante consiste alors à spécifier ces commentaires dans SIB à l'aide du langage de représentation de connaissances Telos. Nous avons présenté à la section 3.4. les principes de spécifications d'objets à l'aide du langage Telos. Nous avons vu qu'il y avait différents niveaux de classification. Les commentaires de mon programme C++ seront alors instanciés au niveau Token (jeton). Le modèle de spécification C++ ainsi que

le modèle de spécification relatif à la classification[10] sont pré-définis au niveau S_Class et se trouvent déjà dans SIB. Au niveau S_Class, pour chaque type commenté, il faut ajouter le type d'attribut **commentaire**. Par exemple, le type **C++Class** aura un type d'attribut **ClassComment** qui sera une instance du type plus général **Comment**. Au niveau Token, le commentaire de notre programme relatif à la classe C++ « Set » sera un objet attribut, instance du type d'attribut **ClassComment** et attribut de la classe Set.

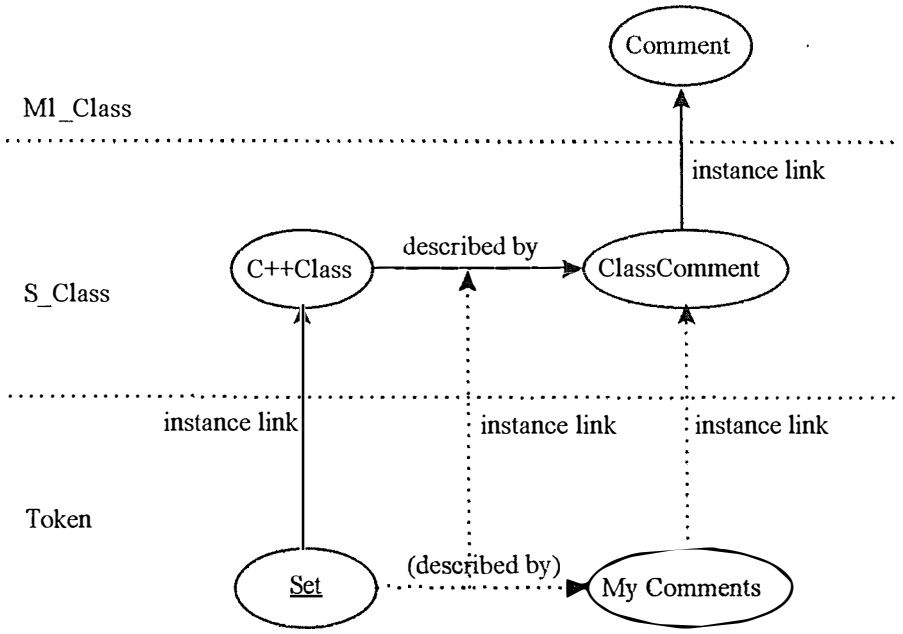

Figure 4.3.: Schéma Telos des spécifications des commentaires.

On obtient ainsi des commentaires spécifiés dans SIB. Le deuxième critère de la documentation en ligne, à savoir **l'organisation**, est atteint. Le troisième élément caractérisant la documentation en ligne, à savoir le **format** (voir 4.3.3.), sera obtenu grâce à l'utilisation du logiciel Static Analyser. Après analyse statique des commentaires de SIB, ce logiciel permettra d'afficher graphiquement ou textuellement les commentaires classifiés.

[10] Les objets que nous allons commenter existent déjà dans SIB et représentent des instances de types C++ et des instances de types relatifs à la classification, types définis préalablement dans SIB au niveau S_Class.

4.3.2. Mots clés.

Concernant la classification de ces commentaires, on va se baser sur la méthode de classification du Static Analyser proposée au chapitre 3. Cependant, comme il ne s'agit pas de classer tous les commentaires des programmes, on va voir comment identifier ceux qui pourraient nous intéresser. On y trouve donc de deux types :

1. Les commentaires sur les catégories classifiables.
2. Autres commentaires publics (par opposition à privés " private ").

Concernant le premier cas, on distingue les commentaires sur les classes et les commentaires sur les méthodes publiques. En effet, les méthodes privées (private) ne sont pas utiles pour la classification puisqu'elles ne sont pas partageables.

Pour les classes, on cherchera tout d'abord à les identifier et ensuite, on pourra y ajouter des commentaires. Pour ce faire, on utilisera les mots clés suivants :

- **CLASS class-name** où class-name est le nom de la classe.
- **CLASSD string** nous permettra d'ajouter une description de cette classe.

Si on désire ajouter des commentaires supplémentaires sur cette classe, on s'intéressera à l'une ou l'autre des informations suivantes :

• à la facette **Abstraction** de cette classe ainsi qu'à un éventuel commentaire sur cette facette.

On notera :
ABSTR abs_name où abs_name est le nom de la facette Abstraction.

ABSTRD string nous permettra d'ajouter une description de cette facette. Le mot anglais "string" signifie qu'on ajoutera une chaîne de caractères (ou plus communément, du texte).

- aux conditions d'existence. On notera **LIVECOND string**. Ceci précise les conditions requises pour l'existence de cette classe. En effet, il y a des cas où une classe dépend de l'existence d'une autre classe.

- aux conditions opérationnelles. On notera **OPERCOND string**. Ceci nous renseigne sur les conditions initiales requises pour que la classe soit opérationnelle.

Concernant les commentaires sur les méthodes, tout comme pour les classes, on cherchera à les identifier mais aussi à y ajouter quelques renseignements supplémentaires. On les identifiera en notant :

-METHOD op_name où op_name (operation name) représente une des opérations effectuées par la classe en question. On voit que ceci fait bien allusion à la facette **OPERATION** de notre modèle de classification. De plus, on peut répéter plusieurs fois ce type de commentaires puisqu'une classe peut contenir plusieurs méthodes.

Les renseignements complémentaires seront de deux types :

- soit la description d'une opération. On notera : **OPERD string.**

- soit les descriptions d'ordre algorithmique. On notera **ALGOD string**. Ici, on pourrait faire allusion à quelques informations un peu plus techniques. Par exemple, une règle d'exception ou une valeur de retour appartiendront aux aspects algorithmiques.

Pour en revenir au second cas, à savoir les autres types de commentaires d'ordre public, on pourrait reprendre diverses informations qui indépendamment, concerneraient les classes ou les méthodes. Bien que cette liste puisse encore être complétée en fonction des besoins, nous avons actuellement considéré les deux types d'informations suivantes:

- les descriptions fonctionnelles. On les notera **FUNCTD string.** Ceci nous renseigne sur les fonctionnalités d'une classe ou d'une méthode. Par exemple, ces informations nous permettrons de savoir à quoi sert la classe qui est décrite.

- les objets alternatifs. On les notera **ALTERNATIVE objects.** Le mot "objects" reprend les différents objets alternatifs. Ce sont des objets (classes ou méthodes) qui lors de la recherche, pourraient intéresser l'utilisateur. Ce critère est assez vague mais dans cette clause, les concepteurs du système d'information introduira les objets qu'il jugera utile. Par exemple, ces objets peuvent agir sur une même structure de donnée ou bien appartenir à la même facette abstraction.

Ceci fut donc la description des différents commentaires qui nous intéressent pour la classification. Cependant, on ajoutera un aspect supplémentaire dénommé **groupe**. On le notera **GROUP { group_body } string.** Dans "group_body", on reprend les différents éléments qui seront regroupés. Chacun d'eux peut contenir des commentaires propres qui seront repris selon la classification proposée précédemment. En fait, l'option "groupe" ne fait qu'englober certains objets et les commenter. Ceci permet d'avoir une certaine souplesse dans l'écriture des commentaires puisque ça nous évite d'écrire plusieurs fois la même chose.

Voici un exemple très simple de ce qui pourrait s'écrire. Il permet d'ajouter un commentaire commun à deux méthodes.

```
GROUP {
    METHOD GetInstof(classe)
```

OPERD Cette fonction retourne les instances d'une classe.

METHOD GetSubClass(classe)

OPERD Cette fonction retourne les sous classes d'une classe donnée.

} Voici les méthodes "Get" d'une classe.

Voici le résumé des types de commentaires proposés précédemment.

Mot clé	Nom complet	Description
POUR	**LES**	**CLASSES**
CLASS	CLASS	Désigne la classe commentée
CLASSD	CLASS DESCRIPTION	Décrit la classe traitée.
ABSTR	ABSTRACTION	Désigne la facette **Abstraction**
ABSTRD	ABSTRACION DESCRIPTION	Décrit la facette **Abstraction**
LIVECOND	LIVE CONDITION	Précise les conditions d'existence
OPERCOND	OPERATIONAL CONDITIONS	Conditions initiales requises
POUR	**LES**	**METHODES**
METHOD	METHODE	Désigne la méthode commentée
OPERD	OPERATION DESRIPTION	Description de la méthode
ALGOD	ALGORITHMICAL DESCRIPTION	Aspects algorithmiques
POUR	**LES CLASSES &**	**LES METHODES**
FUNCTD	FUNCTIONNAL DESCRIPTION	Précise les fonctionnalités
ALTERNATIVE	ALTERNATIVE OBJECTS	Désigne les objets alternatifs
GROUP	GROUP	Permet de regrouper différents objets et de les commenter globalement.

4.3.3. Le format.

4.3.3.1. En entrée.

L'objectif de cette section est de proposer une méthode plus conviviale d'écrire des commentaires. En effet, bien que la solution proposée soit bien intéressante, il faut tout de même tenir compte de certains aspects pratiques. On a vu à la section précédente que lors de la rédaction des commentaires, il faut respecter une certaine syntaxe. Il faut premièrement connaître les mots clés admis, les orthographier correctement et respecter une suite logique entre eux. En quelque sorte, on doit connaître la grammaire de la méthode utilisée.

L'idée est donc d'utiliser un éditeur guidant la syntaxe qui peut directement corriger les erreurs syntaxiques. Lors de la définition d'une classe, l'éditeur reconnaît le mot réservé « class » et ajoute dans le programme des commentaires vides contenant les mots clés adéquats. L'utilisateur n'aura qu'à ajouter les commentaires qu'il juge nécessaires. Ainsi, il n'y aurait plus d'oubli ou de faute d'orthographe pour les mots clés et on éviterait également les fautes grammaticales. L'apport de cette interface servirait en plus à motiver les programmeurs à considérer l'écriture des commentaires comme faisant partie du programme et non pas comme une tâche ennuyeuse supplémentaire.

4.3.3.2. En sortie.

Nous allons présenter les caractéristiques générales de l'interface utilisateur STA ainsi que le résultat de l'analyse statique des commentaires effectuée par ce logiciel. La figure 4.4. présente la fenêtre principale de l'interface utilisateur.

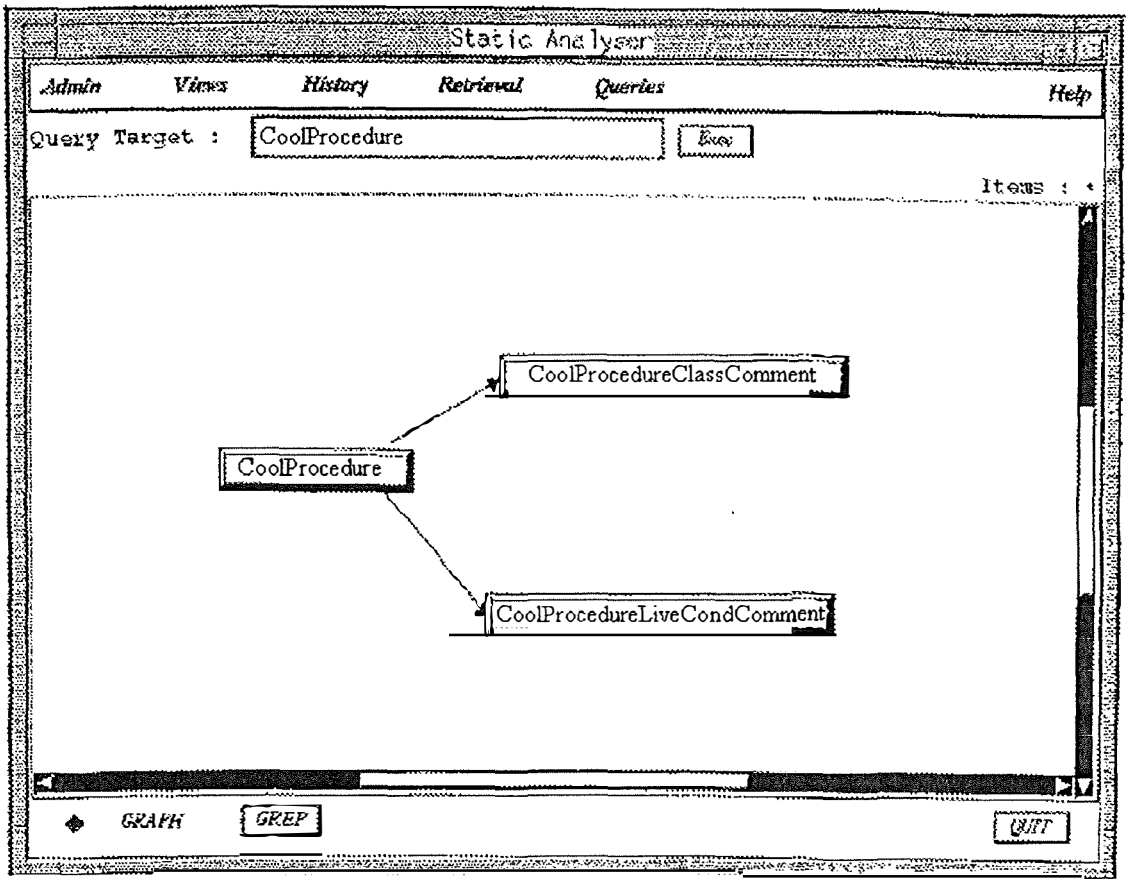

Figure 4.4. : Fenêtre principale de l'interface utilisateur.

C'est une fenêtre Motif standard composée de quatres parties :

- **la barre de menu :**

Les options de base que fournit la barre de menu de l'interface graphique de SIS consiste en six éléments :

Admin menu déroulant contenant des options utilisées pour les configurations générales et pour la communication avec des outils extérieurs.

Views menu déroulant contenant des options utilisées pour le contrôle du mode d'affichage et pour l'exécution des requêtes prédéfinies en mode graphique.

History menu utilisé pour afficher une fenêtre contenant la liste historique des dernières requêtes exécutées.

Retrieval menu utilisé pour afficher des requêtes basées sur des critères de classification.
(exemple : afficher tous les objets dont la facette est « abstraction »).

Queries menu déroulant contenant des options utilisées pour l'exécution des requêtes prédéfinies en mode textuel.

Help menu utilisé pour activer la fenêtre d'aide.

- **les informations sur les requêtes :**

On y trouve principallement la zone *Query Target* qui est un champ d'édition utilisé pour afficher l'objet cible de la requête. On peut entrer le nom d'un objet existant et appuyer sur le bouton *exec* pour exécuter la requête.

- **résultat de requêtes :**

Cette zone est utilisée pour afficher le résultat de la requête. Ce résultat est soit sous forme textuel, soit sous forme graphique. Le résultat graphique d'une requête est affiché sous forme de graphe (de noeuds et de liens).

Dans les deux modes, on peut sélectionner un objet et appuyer sur le bouton gauche de la souris pour générer une nouvelle requête dont la cible est l'objet sélectionné. On peut également appuyer sur le bouton droit de la souris afin d'ouvrir la fenêtre *Object Card* contenant la description textuelle de cet objet.

Cette fenêtre contient les informations complètes sur cet objet. Concernant nos commentaires, chaque type de commentaire est représenté graphiquement par un nom logique (exemple figure 4.4 : CoolProcedureClassComment) et le détail de ce commentaire pourra être consulté en ouvrant la fenêtre *Object Card.* Pour des raisons d'uniformité et de consistance de l'interface utilisateur, les actions de la souris décrites plus haut sont toujours valable ici.

4.4. PARSER.

4.4.1. Spécification.

Entrée : programme source C++.

Sortie : ensemble de commentaires classifiables.

Comme on l'a décrit aux chapitres précédents, le *Parser* permet d'extraire des programmes les caractéristiques des composants classifiables. Il lira un programme source C++ et retirera des commentaires réutilisables. Toute la difficulté de cet exercice est de définir une syntaxe que l'utilisateur devra respecter lors de l'écriture de ses commentaires. Basé sur cette syntaxe, le *Parser* n'aura qu'à identifier les mots clés et extraire les commentaires classifiables. Afin de comprendre cette syntaxe, nous allons présenter à la section suivante la grammaire à respecter. De même, le programme source écrit en langage C se trouve en annexe I.

4.4.2. Grammaire.

Cette grammaire va nous permettre de comprendre avec précision la syntaxe de la méthode proposé.

```
program : program source_code
   | comments program
   ;
source_code:  /* empty */
            | WHOLELINE
            | source_code WHOLELINE
            ;
```

```
comments : "/*" body "*/"
    ;
single_body : classes_comments
            | method_comments
            | group
            ;
body : /* empty */
    | single_body body
    ;
classes_comments : CLASS class_name
                    comments_for_classes
                    comments_for_methods_and_classes
                    | CLASS class_name
                    comments_for_classes
                    ;

class_name : IDENTIFIER
            ;


comments_for_classes : simple_comments_for_classes
                    | simple_comments_for_classes  comments_for_classes
                    ;

simple_comments_for_classes : abstraction_comments
                    | structure_description
                    | live_conditions
                    | operationnal_conditions
                    ;
```

```
abstraction_comments : simple_abstraction_comments
                     | simple_abstraction_comments abstraction_comments
                     ;

simple_abstraction_comments : abstraction_name
                              abstraction_description
                              ;

abstraction_name : ABSTR abs_name
abs_name : IDENTIFIER
         ;

abstraction_description : ABSTRD string
                        ;

live_conditions : LIVECOND string
                ;

operationnal_conditions : OPERCOND string
                        ;

method_comments : METHOD op_name comments_for_methods
                                  comments_for_methods_and_classes
                | METHOD op_name comments_for_methods
                ;

op_name : IDENTIFIER
        ;
```

```
comments_for_methods : operation_description
                     | algorithmic_description
                     ;
comments_for_methods_and_classes : functionnal_description
                     | alternative_objects
                     | functionnal_description alternative_objects
                     | alternative_objects functionnal_description
                     ;
operation_description : OPERD string
                     ;
algorithmic_description : ALGOD string
                     ;
functionnal_description : FUNCTD string
                     ;
alternative_objects : ALTERNATIVE objects
                    ;
objects : object
   | objects object
   ;
object : IDENTIFIER
   ;
group: GROUP '{'group_body '}' group_comments
   ;
group_body : group
           | classes_comments
           | method_comments
           | group_body classes_comments
           | group_body method_comments
           ;
```

```
group_comments : STRING
        ;
string : IDENTIFIER
;
```

4.5. Etudes ultérieures et conclusion.

En considérant que la quantité des informations traitées par et entre les organisations n'arrête pas de croître considérablement, et afin d'éviter qu'à moyen terme, les applications se voient confrontées à des problèmes très complexes dans le traitement de ce flux d'information, la communauté européenne à financé un projet qui consistait à développer un système de support aux applications basés sur des technologies orientées objets. Pour résoudre ce problème, les membres de l'équipe ITHACA ont développé différents outils qui supportent la réutilisation de composants logiciels. La documentation des composants réutilisables s'inscrit donc bien dans cette politique qui consiste à promouvoir la réutilisation, facteur qui s'annonce très important pour l'amélioration de la productivité et de la qualité des logiciels (cfr. 3.1).

Parmi les techniques utilisées, nous avons vu que la réutilisation de composants orientés objet passait par la classification de ces composants (chapitre2). La méthode de classification proposée fut alors le schéma de classification du Static Analyser qui est basée sur des facettes. Les commentaires sont donc classifiés sur base de ce schéma et sont spécifiés dans SIB à l'aide de Telos. L'interface graphique affiche alors le résultat sous forme textuel ou graphique.

Cette analyse pourrait encore être améliorée. On pourrait mieux adapter l'interface en proposant par exemple des requêtes prédéfinies typiques pour les commentaires. Cela nous permettrait de ne plus considérer un commentaire que par rapport à la classe qu'il décrit mais plutôt par rapport à l'ensemble des commentaires.

De telles analyses méritent donc d'être poursuivies et lors de nos programmations futures, elles nous apprennent à prendre suffisamment de recul pour pouvoir situer nos programmes dans une organisation et voir en quoi ils peuvent être réutilisés ultérieurement.

ANNEXE I : Programme source.

```
/* Parser version 1 */
/* Developped by N.Strouboulis */
/* Date : xx/xx/xxxx */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define CLASS              0
#define ABSTR              1
#define ABSTRD             2
#define STRU               3
#define LIVECOND           4
#define OPERCOND           5
#define METHOD             6
#define OPERD              7
#define ALGOD              8
#define FUNCTD             9
#define ALTERNATIVE        10
#define BUF_SIZE           4096
#define MAXCOMMENTS        100
#define BAD                0

int program (char *);
int comments(char *);
int source_code(char *);
int single_comments(char*);
int nextkey(char*, char*);
void init_keywords();
int comment_for_classes(char*);
```

```
int take_comment(char *,char *);

int comment_nbr = 0;
int s_comment_nbr = 0;

char keyword[11][15];

struct COMMENT
{
 char row[BUF_SIZE];
};
struct COMMENT *body[MAXCOMMENTS];

struct SINGLE_COMMENT
{
 char type[20];
 char single_row[BUF_SIZE];
};
struct SINGLE_COMMENT *single_body[MAXCOMMENTS];


main(int argc,char *argv[])
{
FILE *file,*file2;
int i,cpt;
char c;
char s[BUF_SIZE];
int rc;
char output[BUF_SIZE];

init_keywords();
s[0]= '\0';

if (argc < 2)
{
 printf("Parametre requis absent.\n");
 printf("Usage : parser input_file [output_file]\n");
```

```
 exit(1);
}
else if (argc > 3)
 {
 printf("Nombre de parametres incorrect.\n");
 printf("Usage : parser input_file [output_file]");
 exit(1);
 }
file = fopen(argv[1],"r");
if (file == NULL)
{
 printf("Erreur d'ouverture du fichier %s.\n",argv[1]);
 exit(1);
}
if (argc == 2)
 {
 argv[2] = (char *) malloc(11);
 strcpy(argv[2],"output.txt");
 }
file2 = fopen(argv[2],"w+");
if (file2 == NULL)
{
 printf("Erreur d'ouverture du fichier %s.\n",argv[2]);
 exit(1);
}
i = 0;
while (!feof(file) && (i < BUF_SIZE))
{
 c = fgetc (file);
 if ((c == '\n') || (c == '\t'))
      c = ' ';
 s[i] = c;
 s[i+1] = '\0';
 i++;
}
if (i== BUF_SIZE)
{
```

```
 printf("Taille de fichier trop grand : > que %d\n",BUF_SIZE);
 gets(s);
 exit(1);
}
fclose(file);
if (program(s))
{
 for (cpt=0;cpt<s_comment_nbr;cpt++)
 {
 sprintf(output,"%s              %s\n",single_body[cpt]->type,
single_body[cpt]->single_row);
 rc = fputs(output,file2);
 if (rc < 0)
      printf("Erreur d'ecriture dans le fichier output.txt");
 }

 fclose(file2);
 printf("classification OK: s:%s.\n",s);
 gets(s);
 free(body);
}
else
{
 printf("Erreur syntaxique lors de la classification\n");
 gets(s);
}
}

int program(char *s)
{
if (strlen(s) == 0)
 return (1);
if (comments(s))
{
 if (source_code(s))
      if (program(s))
      return (1);
```

```
}
}

int source_code(char  *s)
{
int i, s_size,new_s_size;
char *tmp;

i = 0;
s_size = strlen(s);
tmp = (char*) malloc(s_size);

while ((s[i] != '/' || s[i+1] != '*') && (i < s_size-1))
 i++;

if (i >= s_size-1)
{
 strcpy(s,"");
 return(1);
}
else
{
 new_s_size = s_size - i;
 strncpy(tmp,s+i,new_s_size);
 tmp[new_s_size] = '\0';
 strncpy(s,tmp,new_s_size);
 s[new_s_size] = '\0';
 free(tmp);
 return(1);
}
}
int comments(char *s)
{
int i, comment_size, new_s_size, s_size;
char *tmp;

s_size = strlen(s);
```

```
i = 0;
if ((s[i] == '/') &&  (s[i+1] == '*'))
{
 i=i+2;
 while ((s[i] != '*' || s[i+1] != '/') && (i < s_size - 1))
 i++;

 if (i == s_size -1)
      return(BAD);

 comment_size = i - 2;
 body[comment_nbr] = (struct COMMENT *) malloc(sizeof(struct
COMMENT));
 strncpy(body[comment_nbr]->row,s+2,comment_size);
 body[comment_nbr]->row[comment_size] = '\0';
 if (strlen(body[comment_nbr]->row) > 0)
      if (!single_comments(body[comment_nbr]->row))
           return (BAD);
      else
      {
      comment_nbr++;
      new_s_size = s_size -(i+2);
      tmp = (char*) malloc(new_s_size);
      strncpy(tmp,s+i+2,new_s_size);
      tmp[new_s_size]='\0';
      strncpy(s,tmp,new_s_size);
      s[new_s_size] = '\0';
      free(tmp);
      }
}
return(1);
}
int single_comments(char *row)
{
int pos1, pos2, comment_size;
int init_comment;
char key[15];
```

```
key[0] = '\0';
pos1 = nextkey(row,key);
init_comment = pos1 + strlen(key);
if (strlen(key) != 0)
 /* key found */
 if (strncmp(key,keyword[CLASS],strlen(keyword[CLASS])) == 0)
 {
      /* this is a CLASS comment */
      pos1              =              pos1              +
take_comment(keyword[CLASS],row+init_comment);
      if (pos1 != strlen(row))   /* fin de ligne atteinte */
           pos1 = pos1 + comment_for_classes(row+pos1);
      if (pos1 != strlen(row))   /* fin de ligne atteinte */
           pos1              =              pos1              +
comment_for_methods_and_classes(row+pos1);

      }
 else if
(strncmp(key,keyword[METHOD],strlen(keyword[METHOD])) == 0)
 {
      /* this is a METHOD comment */
      pos1              =              pos1              +
take_comment(keyword[METHOD],row+init_comment);
      if (pos1 != strlen(row))   /* fin de ligne atteinte */
           pos1 = pos1 + comment_for_methods(row+pos1);
      if (pos1 != strlen(row))   /* fin de ligne atteinte */
           pos1              =              pos1              +
comment_for_methods_and_classes(row+pos1);

 }
return(1);
}
```

```
/* nextkey(string)    : initialise key with the the next
classification key (if any) */

int nextkey(char *string, char *key)
{
int s_size = strlen(string);
int i,j;


for (i=0; i < s_size; i++)
 for (j=0; j < 11; j++)
 {
 if ((strncmp(string+i,keyword[j],strlen(keyword[j])) == 0))
      {
      strncpy(key,keyword[j],strlen(keyword[j]));
      key[strlen(keyword[j])] = '\0';
      return i;
      }
 }
return i;
}


int comment_for_classes(char *string)
{
int pos1,i = -1;
char key[15];
key[0] = '\0';
nextkey(string,key);
if (strncmp(key,keyword[ABSTR],strlen(keyword[ABSTR]))==0)
 i = ABSTR;
else                                                    if
(strncmp(key,keyword[ABSTRD],strlen(keyword[ABSTRD]))==0)
 i = ABSTRD;
else if (strncmp(key,keyword[STRU],strlen(keyword[STRU]))==0)
 i = STRU;
```

```
else                                                    if
(strncmp(key,keyword[LIVECOND],strlen(keyword[LIVECOND]))==0)
 i = LIVECOND;
else                                                    if
(strncmp(key,keyword[OPERCOND],strlen(keyword[OPERCOND]))==0)
 i = OPERCOND;
else
 return(0);
if (i != -1)
{
 pos1 = take_comment(keyword[i],string+strlen(keyword[i]));
 if (pos1 != strlen(string))     /* fin de ligne atteinte */
      pos1 = pos1 + comment_for_classes(string+pos1);
}
return(pos1);
}

int comment_for_methods(char *string)
{
int pos1,i = -1;
char key[15];
key[0] = '\0';
nextkey(string,key);
if (strncmp(key,keyword[OPERD],strlen(keyword[OPERD]))==0)
 i = OPERD;
else                                                    if
(strncmp(key,keyword[ALGOD],strlen(keyword[ALGOD]))==0)
 i = ALGOD;
if (i != -1)
{
 pos1 = take_comment(keyword[i],string+strlen(keyword[i]));
 if (pos1 != strlen(string))     /* fin de ligne atteinte */
      pos1 = pos1 + comment_for_methods(string+pos1);
}
return(pos1+strlen(key));
}
```

```
int comment_for_methods_and_classes(char *string)
{
int pos1,i= -1;
char key[15];
key[0] = '\0';
nextkey(string,key);
if (strncmp(key,keyword[FUNCTD],strlen(keyword[FUNCTD]))==0)
 i = FUNCTD;
else                                                    if
(strncmp(key,keyword[ALTERNATIVE],strlen(keyword[ALTERNATIVE])
)==0)
 i = ALTERNATIVE;
if (i != -1)
{
 pos1 = take_comment(keyword[i],string+strlen(keyword[i]));
 if (pos1 != strlen(string))     /* fin de ligne atteinte */
      pos1              =              pos1              +
comment_for_methods_and_classes(string+pos1);
}
return(pos1+strlen(key));
}

int take_comment(char *key, char *string)
{
int init_key2, comment_size;
char key2[15];

key2[0] = '\0';
comment_size = nextkey(string,key2);
if (strlen(key2) == 0)
 comment_size = strlen(string);

single_body[s_comment_nbr]   =   (struct   SINGLE_COMMENT   *)
malloc(sizeof(struct SINGLE_COMMENT));

strncpy(single_body[s_comment_nbr]->type,key,strlen(key));
```

```
single_body[s_comment_nbr]->type[strlen(key)] = '\0';
strncpy(single_body[s_comment_nbr]->single_row,
string,comment_size);
single_body[s_comment_nbr]->single_row[comment_size] = '\0';
s_comment_nbr ++;
return(comment_size + strlen(key));
}
   /* init_keyword : initialise keyword table */
   void init_keywords()
   {
   strncpy(keyword[CLASS],"##CLASS##",9);
   keyword[CLASS][9] = '\0';

   strncpy(keyword[ABSTR],"##ABSTR##",9);
   keyword[ABSTR][9] = '\0';

   strncpy(keyword[ABSTRD],"##ABSTRD##",10);
   keyword[ABSTRD][10] = '\0';

   strncpy(keyword[STRU],"##STRU##",8);
   keyword[STRU][8] = '\0';

   strncpy(keyword[LIVECOND],"##LIVECOND##",12);
   keyword[LIVECOND][12] = '\0';

   strncpy(keyword[OPERCOND],"##OPERCOND##",12);
   keyword[OPERCOND][12] = '\0';

   strncpy(keyword[METHOD],"##METHOD##",10);
   keyword[METHOD][10] = '\0';

   strncpy(keyword[OPERD],"##OPERD##",9);
   keyword[OPERD][9] = '\0';

   strncpy(keyword[ALGOD],"##ALGOD##",9);
   keyword[ALGOD][9] = '\0';
```

```
strncpy(keyword[FUNCTD],"##FUNCTD##",10);
keyword[FUNCTD][10] = '\0';

strncpy(keyword[ALTERNATIVE],"##ALTERNATIVE##",9);
keyword[ALTERNATIVE][9] = '\0';
}_
```

ANNEXE II : Exemple de classification de commentaires.

Ci-dessous, nous allons présenter un exemple concret en classant les commentaires d'un composant C++ "telos_object". Le programme proposé est repris d'un fichier "header" (c'est-à-dire un fichier qui contient la définition de certaines objets ou classes) utilisé pour décrire les objets du langage de représentation des connaissances : TELOS.

L'objectif était alors de modifier la structure des commentaires en y ajoutant les mots clés adéquats. Le **parser** pourra alors facilement distinguer les commentaires utiles des autres qui ne seront pas réutilisés. Il est bien évident qu'au préalable, une analyse a été faite pour décrire la classification de tous ces objets. Par exemple, si on veut ajouter des commentaires sur la facette **abstraction** d'une classe, il est bien évident qu'il faut préalable savoir que cette classe appartient à cette facette.

```
/*****************************************/
/*                                       */
/*      TELOS C++ Version                */
/*      File : object.h                  */
/*      Author : Martin Doerr            */
/*      Created : 14/12/90               */
/*                                       */
/*****************************************/
#include <stdlib.h>
#include <stdio.h>
#include "objects_typ.h"
#include "objects_sizes.h"
/*

* We repeat the objects sysid in the extension for
* redundancy.
* The extension may be used to hold sysid's or strings.
* The extensions of one object form are simply linked list.
* The linf pointers hold memory addresses in the case that
* they are loaded to memory, whereas when written to the
```

```
* disk they hold disk addresses, it means numbers of bytes

* from the beginning of the databases ~.obj file.
*
*/

class sys_cat;
class telos_oext {
  public:
  SYSID       o_sysid;      /* the objects sysyid */
  telos_oext *o_ext_next;  /* disk/memory address of next extension */
  union {
  char  o_char[TLS_EXT_SZ*sizeof(int)];
  int   o_id[TLS_EXT_SZ];
  };
  telos oext(SYSID id) {o_sysid = id;o_ext_next = NULL;};
};

/* CLASS telos_object
  CLASSD Telos_object is the superclass for the special classes in
which the telos objects are stored. telos objects are described by
three sets :
- the set of classes it is an instance of
- the set of its superclasses if it is a class
- the set of its attributes
attributes are objects by themselves which have two additional fields
:
- the  "from" objects where it is defined in
- the "to" object which is the attribute value itself
  ABSTR Data_Object
  STRU All references to objects in the above mentioned sets and in
the "to" and "from" fields are done by system identifiers. All these
sets (fields) are accompanied by the corresponding  inverse link sets
(i.e. : "from" <-> "link", "to" <-> "link by", "isa" <-> "subclass",
"inst_of" <-> "inst_by"), except for references to bulletin system
classes.
*/
```

```
/* * Le commentaire qui suit ne sera pas classer pour une   *
éventuelle réutilisation puisque ce sont des informations * d'ordre
privé.
*/
/*
* All these sets are implemented by fixed size array ("slots")
* as a starting point and are extended dynamically into
* extensions which are not further structured or specialized.   *
These fixed size arrays should * be later adjusted to the     *
average space needed, to avoid extensive use * of extensions  * and
waste of disk/memory space.
* Format of the slots :
* The last sysid carries a sentinel bit (SYSID_END). An absolute *
zero is invalid before the last sysid and indicated a         *
corrupted database or programming error. If the set is        *
continued beyond the end of the
* slot's array or beyond a contiguous field of elements of      *
arbitrary
* size within an extension, the last element of the array or the
* contiguous field  becomes an "continuation" address, which is * a
virtual address, which over the space of all the extensions * of the
object, indicating the storage location of the next    * element of
the set. The continuation address carries the      * "SYSID_CONT" bit.
Hence in
* the current implementation sysids are restricted to 30 bits.
* the decision was made to allow for slot arrays of size 1, in  * the
case * we mostly do not have more elements (e.g. inst_of, * isa, link
of attribute classes).
* The decision to use virtual addresses was made to avoid
* updating all
* these addresses when loading an object from disk.
* epending on the order during data input, the contents of one
* slot may * be highly fragmented over the object's extensions, *
requiring unnecessary space for continuation addresses and
* unnecessary jumps during retrieval. This is why the contents  * of
an object having extensions should be compressed with the  * "copy"
method" before saving * on disk.
*/
```

```
class telos_object {
/*
* private declarations
*/
/* find the position to put a sysid into any slot                    */
int find_id(SYSID  **slot, SYSID  **lim, SYSID id , int  insType);

 /* find the sysid to be deleted in any slot                  */
int find_iddel (SYSID **slot, SYSID **lim,SYSID id);

/* after "find_id", shift  the rest of the slots contents up  */
shift_up_id (SYSID *slot, SYSID *lim, int flag, SYSID id, int onsType)

/* after "find_iddel", shift  the rest of the slots contents down   */
shift_down_id (SYSID *slot, SYSID *lim);

/* copy content of any slot slot0 to an array(!) slot1 until*/
/*end of contents of slot0 or limit slim in slot1.*/
int copy_id(SYSID  **slot0, SYSID *slot1,SYSID *slim)

/* allocate a new free space  on the extension space */
SYSID *alloc_ext(int flag,SYSID *sp, SYSID *lim, int insType);

protected :

   void mark_last_type(SYSID *sp, int insType);
   int insertAfterLast5SYSID id);
   /* put a SYSID to any slot of a telos object  : */
int generic_putsys(SYSID *slot, SYSID *slim, SYSID id, int insType);

   /* get all sysid's from any slot of a telos object:  */
int getsys_by_copy(SYSID *slot, SSET *set);
SYSID_SET *generic_getsys(v *slot, SYSID_SET *set);

/* delete a sysid from any slot of telos object */
void generic_delsys(SYSID *slot, SYSID *slim, SYSID id);

   /* copy contents of a slot  slot0 of objects op completly to     */
/* slot1 of the calling object  (uses copy_id)          */
```

```
int trans_id (SYSID *slot0, SYSID *slot1, SYSID *slim, telos_object
*op);

void very_new_trans_id(SYSID *slot0, SYSID *slot1, SYSID *slim,
telos_object *op);

/* using the extension -relative virtual address "ca", get a pointer
*/
/* into the corresponding extension and its last element *lim
*/
  SYSID *get_cont(int ca, SYSID **lim);

/* get and link a new extension */
  telos_oext *get_ext();

  /* deallocate all extensions of the object calling    */
void del_ext();

/* print the contents of any slot for debugging         */
void print_id(SYSID *slot);

/* print the contents of any slot for debugging         */
int stat_id (SYSID *slot);

/* print the contents of the (common) superclasses part of debugging
*/
void print_com();
  telos_oext  *lastExt;
  int   lastMarkerType;

public:
SYSID   o_susid;          /* the objects sysid */
telos_oext    *o_ext;     /* address of extension */
short   o_type;           /* object TYpe*/
short   o_flag;           /* object flag */
int     o_size;           /* size overall extensions */

  void set_lastExt(telos_oext *ptr) {lastExt = ptr; }
  void  reset_markers() {
        lastExt = (telos_oext *) 0;
```

```
        lastMarkerType = 0;
        }
telos_object() {reset_markers();}
/* GROUP {
METHOD isUnresolved
METHOD putResolved
METHOD putUnresolved
} Handle the "unresolved object" flag during data input.
*/
isUnresolved() {return (o_flag&O_UNRESOLVED);};
void putResolved() {o_flag &= ~O_UNRESOLVED;};
void putUnresolved() {o_flag = O_UNRESOLVED;};

IsExplicit() { return (o_flag&O_EXPLICIT_LEVEL);};
void putDeduced() {o_flag&= ~O_EXPLICIT_LEVEL;};
void putExplicit() {o_flag = O_EXPLICIT_LEVEL;};

/*METHOD getSysIsa
  OPERD       Get all System class I am ISA of
              Token throughM4_class, nodes and links separated
*/
virtual SYSID_SET  *getSysIsa(SYSID_SET  *set, sys_cat*);

/*METHOD getSysInstof
  OPERD       Get all System class I am instance of
              Token throughM4_class, nodes and links separated
*/




virtual SYSID_SET  *getSysInstOf(SYSID_SET  *set, sys_cat*);

/*METHOD      getSysClass
  OPERD       Get the level-systeme class I am instance of   Token
throughM4_class, nodes and links not separated
*/
```

```
virtual SYSID  getSysClass();

/*METHOD      d_belongs_to
  OPERD Am I member of this set of classes? In case of a System Class,
        it answers yes if a IsA relation holds to one of these
*/
virtual int  d_belongs_to(SYSID_SET  *set, sys_cat*) {
        return (set->set_member_of(o_sysid));};
/* continued    (...) */
```

BIBLIOGRAPHIE :

[APE92] C++ Extended Library, APEX 1.0 Information brochure,1992..

[Booch91] Grady Booch, "Object Oriented Design with Applications", Benjamin/Cummings, 1991

[CDV93] Panos Constantopoulos, Martin Doerr and Yannis Vassiliou, " Repositories for Software Reuse : The Software Information Base", ITHACA Technical Overviews,1993

[CONS92] Constantopoulos, P., Jarke, M. Mylopoulos, J., and Vassiliou, Y., The Software Information Base: A Server for Reuse, ITHACA.FORTH.92.E2.#1, Institute of Computer Science, Foundation of Research and Technology - Hellas, January 1992.

[DK93] Martin Dorr, Polivios Klimathianakis, " A Telos Model for C++ Programs Static Analysis Data ", Institut of Computer Science, Foundation of Research and Technology - Hellas, Working Note #9, Novembre 1993.

[DKP94] Daskalakis D., Karamaounas P. and Prekas N., "SIS Graphical Analysis Interface, User's Manual, SIS Static Analyser", Runtime System, Institut of Computer Science, Foundation of Research and Technology - Hellas, 1994.

[DOER93] Doerr M. and Petra E., "Classifying C++ Reusable Components", Institut of Computer Science, Foundation of Research and Technology - Hellas, 1993.

[ITHACA92] ITHACA.FORTH.92.E2.#2, Implementation of the SIB System

[KARL92] Karlsson E-A., Sorumgard S. and Tryggeseth E., "Classification of Object-Oriented Components for reuse", Division of Computer Systems and Telematics, Norwegian Institut of Technology, January 1992.

[KMSB89] Koubarakis M., Mylopoulos J., Stanley M. and Borgida A., "Telos Features and Formalization", Institut of Computer Science Forth, Technical Report FORTH/CSI/TR/1989/018, Feb. 1989 also University of Toronto, Computer Science Dept., Technical Report KRR-TR-89-4, Feb. 1989.

[PDF87] Prieto-Diaz R. and Freeman P., "Classifying software for reusability", IEEE Software, pp. 6-16, January 1987.

[Shirk88] Shirk, Henrietta Nickels., " Technical Writers as Computer Scientists : The Challenges of Online Documentation.", The MIT Press, pp. 311-327, 1988.

[Walker88] Walker, Janet H., "Supporting Document Development with Concordia", IEEE Computer, pp. 48-59, January 1988.

[Wood88] Wood, M. and Sommerville, " I. An information system for software components", SIGIR Forum 22,3 (Spring/Summer 1988) , 11-25

[Yeorg94] Yeorgiannakis George, " A Storage and Memory Management Mechanism for Objects in Telos", Master of Science Thesis, Department of Computer Science, University of Crete, Février 1994.